Generera användargränssnitt baserade på JSON Scheman

JULIUS RECEP COLLIANDER CELIK

Civilingenjör Informationsteknik
Datum: 3 maj 2018
Handledare: Patric Dahlqvist
Examinator: Anders Västberg
Uppdragsgivare: LS Elektronik AB
Engelsk titel: Generating user interfaces based on JSON Schemas
Skolan för elektroteknik och datavetenskap

Innehåll

Kapitel 1

Introduktion

Vad jag känner till är detta det första publikt dokumenterade försöket av att generera ett användargränssnitt från JSON Schema, som är implementerat i något annat än HTML, CSS eller JavaScript. Detta är 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.

Examensarbetet handlade om att undersöka möjligheten att skapa en modell för att beskriva och annotera redigerbar data, och sedan automatiskt generera ett användargränssnitt. Det här kapitlet introducerar hela arbetet. Kapitel 1.1 ger en bakgrund till arbetet, vilket innefattar både en enkel teoretisk bakgrund, samt en bakgrund till företaget arbetet utfördes hos, samt systemet som arbetet utvecklades mot. Kapitel 1.2 diskuterar systemet arbetet utvecklades mot i större detalj. Kapitel 1.3 presenterar problemet med en frågeställning. Kapitel **??** föreslår en hypotes som arbetet grundar sig på. Kapitel 1.4 och 1.5 diskuterar syftet och målet med arbetet. Riskerna diskuteras i Kapitel 1.6. Kapitel 1.7 presenterar metodvalet. Kapitel 1.8 beskriver arbetets omfattning. Resten av rapportens disposition presenteras i kapitel 1.9.

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. Kapitel 1.1.1 beskriver JSON och JSON Schema vilket är en

stor del av teknikerna som arbetet undersökte. Kapitel 1.1.2 beskriver företaget arbetet utfördes hos, samt 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 LS Elektronik AB *(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 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 SoftRadio. 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, som beskrevs i 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 ett 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 framti-

da 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 tillräckligt generellt sätt att administratörsprogrammet var framtidssäkert för framtida version av Mimer.

Problemet kan förenklat delas up i två problem. Ena problemet är att olika operatörsdatorer har olika uppsättningar konfigurationer att konfigurera, vilket är anledningen till att användargränsnittet måste anpassas beroende på operatörsdatorn den kopplar upp sig mot. För att lösa det problemet skulle möjligtvis användargränsnittet kunna extrapolera information från konfigurationsfilen, som sparar inställningar på operatörsdatorn, och därmed lista ut hur olika inställningar kan redigeras. Det diskuteras mer i kapitel **??**. Anledningen till att en sådan lösning inte är lämplig är delvis för att den inte skulle kunna vara felfri, vilket diskuteras i kapitel **??** men också för att konfigurationsfilen innehåller inställningar som administratörsklienten inte ska kunna konfigurera. Det andra problemet är därför att LSE vill kunna bestämma vilka inställningar administratörsklienten möjliggör konfiguration av. Därför måste det finnas en fil, antingen hårdkodad eller genererad, på serverdatorn som bestämmer hur användargränsnittet på klientdatorn ska se ut, och hur klienten får manipulera datan på serverdatorn.

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 kun-

na konfigurera inställningar hos olika datorer med olika versioner av Mimer SoftRadio, 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 SoftRadio att enkelt kunna konfigurera inställningar, oavsett version eller uppsättning extra funktionalitet. Det skulle kunna innebära att Mimer SoftRadio 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.

1.5.1 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 SoftRadio används 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 SoftRadio. 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.

1.6 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.

1.7 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?

1.8 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.

1.9 Disposition

Kapitel 2 presenterar den teoretiska bakgrunden.

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 beskrivs. Kapitel 2.4 diskuterar användningsområden av JSON Schema samt listar kända implementationer.

2.1 JSON

JSON erbjuder stöd för några enkla datastrukturer: textsträngar *(string)* (`"hej värld"`), tal *(number)* (`4`), ett tomt värde *(null)* (**`null`**), samt booleska värden *(booleans)* (**`false`**). JSON erbjuder dessutom stöd för två komplexa datatyper vilket är vektorer *(array)*, en ordnad lista av JSON-värden vilket visas i Figur 2.1, samt objekt *(object)*, vilket är en oordnad mängd av namn-värde-par *(properties)*, som visas i Figur 2.2. Resten av rapporten kommer utbytbart använda JSON-värde, JSON-data, JSON-fil och JSON-dokument för att förklara en av de sex datastrukturerna 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.

```
["hej värld", 4, null, false]
```

Figur 2.1: Exempel på JSON-array

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

Figur 2.2: Exempel på JSON-object

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. En hypotetisk förfrågan till en webbtjänst skulle kunna se ut som i Figur 2.3, där en klient förfrågar om de nuvarande väderförhållandena i Stockholm i Sverige. Svaret från webbservern skulle kunna se ut som i Figur 2.4 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 skulle kunna delvis vara en förklaring för populariteten.

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

Figur 2.3: Exempel på förfrågan till webbserver

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

Figur 2.4: Exempel på svar på förfrågan från 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 2.2 skulle kunna valideras enligt JSON Schemat som visas i Figur 2.5. 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 va-

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

Figur 2.5: Exempel på simpelt JSON Schema

lidering 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 nyckelord som `"title"` och `"description"`, där `"title"` är en kort förklaring för JSON värdet den validerar, och `"description"` är en längre förklaring. [5]

2.3.3 JSON Schema Hyper-Schema

JSON Schema skapas till stor del för användandet hos 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]

2.4 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 [7]:

- 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. [7]

Implementationerna som genererar JSON Scheman från JSON data:

- Schema Guru *(Scala)* [8]
- JSON Schema Generator *(Visual Studio)* [9]
- json-schema-generator *(JavaScript / JSON)* [10]

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

- Json.NET Schema *.NET* [11]
- NJsonSchema for .NET *.NET* [12]
- typescript-json-schema *(TypeScript)* [13]
- Typson *(TypeScript)* [14]

Implementationerna som genererar JSON Scheman från andra liknande beskrivningar:

- Liform *(PHP)* [15]
- JSL *(Python)* [10]
- JSONSchema.net *(Online webbverktyg)* [14]
- Schema Guru Web UI **Obs:** Verktyget hittades ej och kommer därför exkluderas från resten av rapporten.
- APIAddIn *(Sparx Enterprise Architect)* [16]

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* [17]
- jsonCodeGen *Groovy* [18]
- aeson-schema *Haskell* [19]
- AutoParse *Ruby* [20]
- json-schema-codegen *Scala* [21]
- Argus *Scala* [22]
- Bric-à-brac *Swift* [23]
- gojsonschema *Golang* **Obs:** Verktyget saknade information på engelska eller svenska och kommer därför exkluderas från resten av rapporten. [24]
- jsonschema *Golang* [25]

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 [26]
- Angular Schema Form [27]
- Angular2 Schema Form [28]
- JSON Editor [29]
- JSON Form [30]
- json-forms [31]
- JSONForms [32]
- Jsonary **OBS!**
- liform-react [33]
- Metawidget [34]
- pure-form webcomponent **Obs not found**
- React JSON Schema Form [35]
- React Schema Form [38]

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 3.2.

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 3.3. 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

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

Figur 3.1: Exempel på enkelt json objekt

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    }
  }
}
```

Figur 3.2: Exempel på genererat schema

skulle kunna vara det booleska värdet *false*. Då skulle schemat se ut som i figur 3.4.

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 3.5. Alla exempel är giltiga scheman till objektet i det givna exemplet. Det finns ingen möjlighet att förstå vilket schema 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.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "pattern": "^[A-Z].*$"
    }
  }
}
```

Figur 3.3: Exempel på genererat schema med strängmönster

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "name": {
      "enum": ["Julius", "Erik", false]
    }
  }
}
```

Figur 3.4: Exempel på genererat schema med *enums*

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    }
  },
  "required": ["name"]
}
```

Figur 3.5: Exempel på genererat schema med dolda *properties*

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. [11]–[14]

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. [10], [15]

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. [16]

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. [36] 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. [37]

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. [17]

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. [18]

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. [19]

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. [20]

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. [21]

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. [22] Bric-à-brac är också ett verktyg som liknade Argus i funktionalitet, fast implementerat i språket Swift [23]. En notering är att dokumentationen är motstridig då den påstår att verktyget erbjuder en oföränderlig *immutable* datastruktur, men den i exempelkoden visar motsatsen [23]. 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. [24]

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. [25]

Många av dessa projekt saknar tillräcklig dokumentation, vilket gör det omöjligt att använda dem i ett projekt. Dessutom är parsning-

en 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 implemen-

tationen 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. [26]

Andra lösningar som utökade det oficiella JSON Schemat är:

- Angular2 Schema Form [28]
- JSON Editor [29]
- json-forms [31]
- JSONForms [32]
- liform-react [33]
- Metawidget [34]
- React JSON Schema Form [35]

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 [27]
- Angular2 Schema Form [28]
- JSON Form [30]
- json-forms [31]
- JSONForms [32]
- liform-react [33]
- React JSON Schema Form [35]
- React Schema Form [38]

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

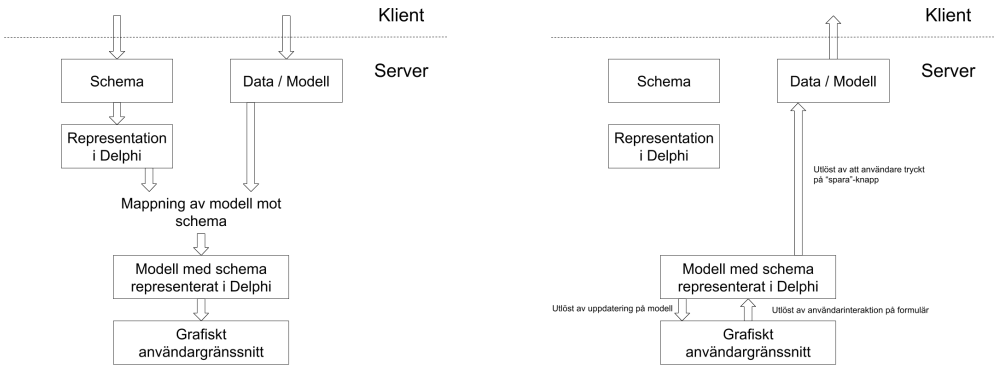
Arbetet kan del

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*. [11]–[14] Delphi erbjuder liknande funktionalitet med *Attributes (RTTI)* [39]. 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. [10] 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 genererat 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 ex-

akt 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 *enums* vilket diskuteras mer i kapitel 4.4.4. 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

4.4.2 Textsträngar och *format* i användargränssnittet

4.4.3 Booleska värden och heltal i användargränssnittet

4.4.4 Flervalsalternativ och *enums* i användargränssnittet

Kapitel 5

Diskussion, slutsats och fortsatt arbete

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[36] H. Andrews, A. Wright och Internet Engineering Task Force, "Draft-wright-json-schema-validation-01", tekn. rapport, 2017. URL: https://tools.ietf.org/html/draft-wright-json-schema-validation-01.

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

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

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

Bilaga A

API-beskrivning

TODO!