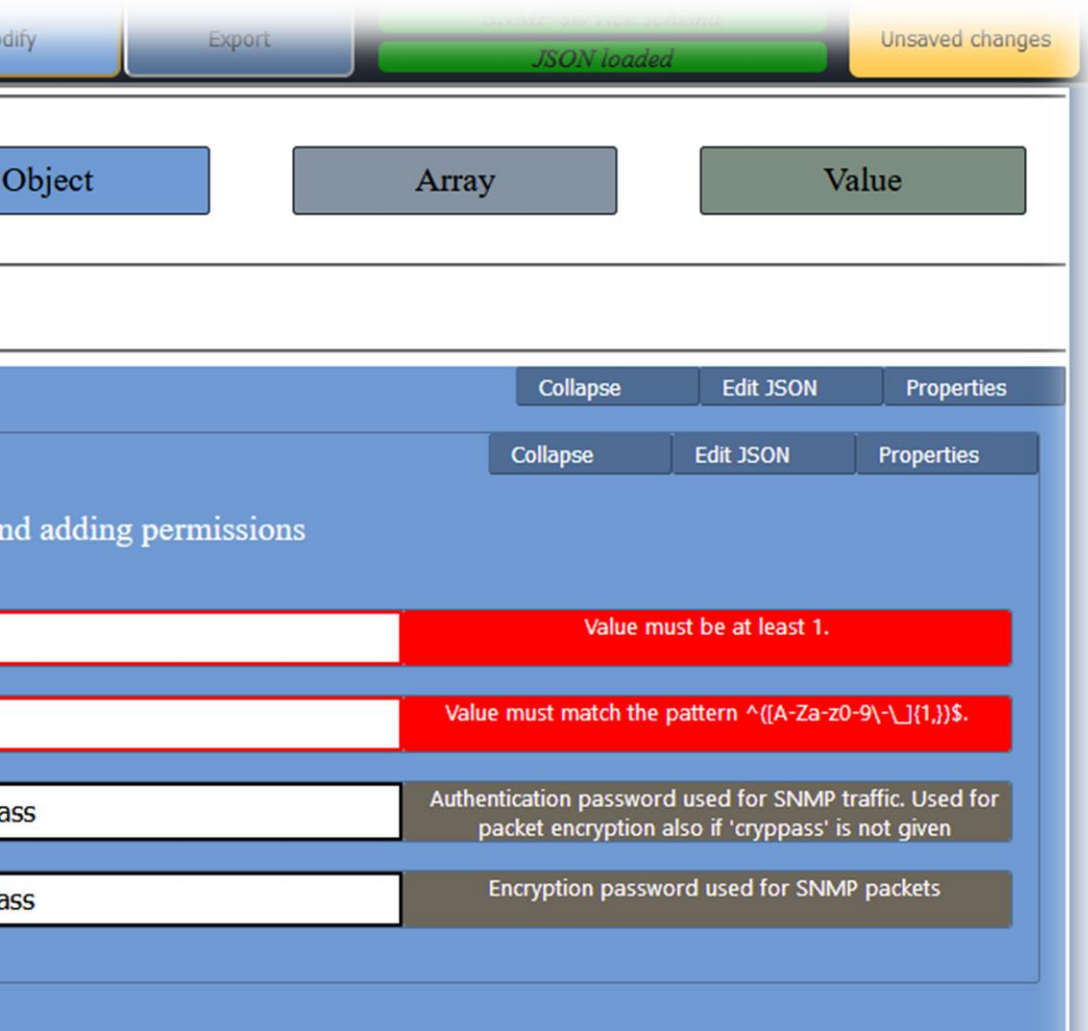
Juho Kaartinen

Extended JSON Configuration Tool – konfiguraatiotyökalun suunnittelu sekä toteutus



Insinööri, AMK

Tieto- ja viestintätekniikka

Kevät 2019

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Kaartinen, Juho

Työn nimi: Extended JSON Configuration Tool –konfiguraatiotyökalun suunnittelu sekä toteutus

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: JSON, schema, konfigurointi, varmennus, validointi

Opinnäytetyö toteutettiin toimeksiantona Bittium Wireless Oy:lle. Toimeksiantaja suunnittelee ja valmistaa mobiiliverkkojen muodostamiseen tarkoitettuja ohjelmistoradioita sekä reitittimiä. Opinnäytetyössä tutkittiin mahdollisuuksia varmentaa JSON-formaattia käyttävien laitekonfiguraatioiden sisältö sekä toteutettiin varmennusta konfiguraatiotiedostojen teossa hyödyntävä työkalu sen ensimmäiseen vaiheeseen. Työkalun pääasiallisena tarkoituksena on estää virheellisten arvojen päätyminen ohjelmistoradioiden konfiguraatioihin sekä konfiguraatioiden laatimisprosessin virtaviivaistaminen. Osassa toimeksiantajan valmistamissa laitteissa on jo sisäisesti toteutettu konfiguraation varmennus, nyt sama ominaisuus haluttiin tuoda jo konfiguraation tekovaiheeseen.

Aluksi laadittiin karkea aikataulu opinnäytetyön kirjallisen osuuden toteutuksesta sekä tehtiin työkalun ohjelmistoarkkitehtuurisuunnitelma toimeksiantajan sille kohdistamien vaatimusten pohjalta. Vaatimusten mukaan työkalulla tuli olla mahdollista muokata JSON-muotoisia konfiguraatiotiedostoja sekä varmentaa niitä JSON Schema –standardin mukaisia malleja vastaan. Standardi julistaa joukon avainsanoja, joiden pohjalta mallit implementoiva ohjelma pystyy estämään virheelliset syötteet. Työkalulla täytyi myös pystyä luomaan uusia konfiguraatiotiedostoja mallien pohjalta. Vaatimusten pohjalta laadittiin alustavat käyttötapaukset. Lisäksi kartoitettiin jo olemassa olevia kirjastoja, jotka implementoivat JSON Schema –standardin mukaisen varmennuksen. Kartoituksen perusteella löytyi käyttökelpoinen, avoimen lähdekoodin lisenssillä julkaistu kirjasto, joka otettiin käyttöön työkalun implementointivaiheessa. Käyttöliittymän alustavan rakenteen suunnitelma muodostettiin käyttötapausten pohjalta.

JSON Schema –standardia tutkittiin sillä laajuudella, että varmistettiin riittävä ymmärrys sen rakenteesta ja käytöstä sekä saatiin koottua esimerkkitapaukset kaikista yleisistä ja oletettavasti työkalun implementoinnissa vastaan tulevista käyttötapauksista. Tutkimuksen laajuuden rajauksessa käytettiin apuna aiemmin kertynyttä osaamista laitteiden konfiguraatioista. Standardin tutkimisesta saatu tieto koottiin oman kappaleen alle itsenäiseksi kokonaisuudeksi, jotta sitä voidaan tarvittaessa hyödyntää toimeksiantajan sisäisessä koulutuksessa sekä perehdytyksessä.

Työkalu toteutettiin verkkoselaimen päällä toimivaksi ohjelmistoksi. Näin saavutettiin tilanne, jossa työkalun käyttö on käyttöjärjestelmästä riippumatonta ja vaatii ainoastaan modernin verkkoselaimen toimiakseen. Työkalu toteutettiin Javascript-ohjelmointikielellä sekä HTML- ja CSS-kuvauskielillä. Työkalun koodipohja jaettiin liiketoimintakerrokseen sekä käyttöliittymään. Liiketoimintakerros paljastettiin käyttöliittymälle ohjelmointirajapinnan kautta. Käyttöliittymä toteutettiin yhdelle HTML-kuvaustiedostolle. Käyttöliittymän näkymää muokataan dynaamisesti Javascript-koodilla käyttäjän toiminnan mukaan.

Opinnäytetyön tuloksena syntyi ohjelmisto, joka täyttää toimeksiantajan vaatimukset. Työkalulla pystyy laatimaan sekä varmentamaan JSON-muotoisia konfiguraatiotiedostoja selkeän käyttöliittymän kautta. Työkalu on tarkoitus saattaa jatkokehityksen piiriin. Jatkokehityksen kohteita kartoitettiin ensimmäisen vaiheen tavoitteiden täytyttyä.

KAMK • University of Applied Sciences

Abstract

Author: Kaartinen, Juho

Title of the Publication: Design and Implementation of the Extended JSON Configuration Tool

Degree Title: Bachelor of Engineering, Information and communication technology

Keywords: JSON, schema, configuration, validation

This thesis was made as a commissioned assignment for Bittium Wireless Ltd. The company designs and manufactures routers and software-defined radios used to form mobile ad-hoc networks. In the thesis, different possibilities for the creation and validation of JSON-formatted configurations were studied and a tool capable of these features was implemented to its first phase. The tool's main purpose is to prohibit faulty values from reaching the final configuration and to streamline the process of creating configurations. Some of the company's products already internally implement the configuration validation. The tool made during this thesis process aimed to bring that validation to the configuration creation phase.

The process started with a composition of a crude timetable for the textual part of the thesis and a creation of a rudimentary software architecture for the tool based on the requirements given by the commissioner. Requirements declared that the tool should be capable of modifying JSON-formatted configuration files and validating them against models following the JSON Schema –standard. The standard declares a group of keywords which allow the program implementing the standard to prevent faulty input. The tool also had to be capable of creating configuration files using schema models as templates. Based on these requirements, the initial use cases were formed. Third-party libraries implementing the JSON Schema–standard were also surveyed. This led to finding a useful, open-source licensed library which was taken into use under the implementation phase. The user interface's structure was planned to a preliminary level using the use cases as a basis.

JSON Schema –standard was studied to a level which allowed to achieve a proper understanding of its structure and usage and to gather example cases of the standard's common keywords and use cases. Previous knowledge of the configuration process was used to limit the depth of the study process. Knowledge and examples gathered during the thesis process were collected under a single title in the thesis to allow their use in the commissioner's internal trainings and introductions if needed.

The tool was implemented to use web browsers as the platform. This resulted in a situation where the tool is basically operating system agnostic, needing only a modern browser to work. The tool was written with Javascript –programming language and HTML and CSS markup languages. The tool's code base was divided to two layers, one containing the business logic and the other containing the user interface. Business logic was exposed to the user interface via an application programming interface. The user interface was implemented on a single HTML-file, using Javascript to dynamically change the file's content depending on the user's actions.

As the result of the thesis process, a usable tool meeting the commissioner's requirements was created. It is capable of creating and validating JSON-formatted configuration files via a distinct user interface. The tool is meant to be further developed in the future. After the first phase, as implementation was achieved, future development goals were mapped.

Sisällys

Symboliluettelo

API	Application Programming Interface. Ohjelmointirajapinta, jonka kautta ohjelmiston toimintaa voidaan muuttaa.
IETF	Internet Engineering Task Force. Internetissä käytettäviä standardeja valvova sekä ylläpitävä toimielin.
Internet-Draft, I-D	IETF:n julkaisema dokumentti, joka sisältää alustavan teknologisen määritelmän, tutkimuksen tuloksen tai muuta teknistä informaatiota. Voimassa puoli vuotta kerrallaan.
JSON Schema	JSON-formaattia noudattava validointiformaatti, jonka avulla JSON-tiedoston arvoja voidaan rajata ja varmentaa.
Liiketoimintakerros	Ohjelmiston idean ja/tai palvelun toteuttava osa ohjelmistosta, puhekielessä yleisesti bisneslogiikka.
MANET	Mobile ad-hoc network. Langaton, tiettyä tarkoitusta varten muodostettu verkko, jossa ei ole erillisiä tukiasemia.
MIME	Multipurpose Internet Mail Extension. Standardi, joka ilmaisee, mitä formaattia tiedosto tai data on.
regex, regular expression	Säännöllinen lauseke. Sarja merkkejä, jotka muodostavat hakulausekkeen. Käytetään merkkijonojen analysoinnissa ja manipuloinnissa.
SPA	Single page application. Web-sovellus, jossa web-sivua muokataan dynaamisesti käyttäjän selaimella sen sijaan, että uudet sivut ladattaisiin serveriltä.
TAC WIN	Tactical Wireless IP Network. Bittium Oyj:n tuoteperhe, jonka avulla voidaan luoda langattomia paikkariippumattomia tietoverkkoja.
URI	Uniform Resource Identifier. Merkkijono, joka yksiselitteisesti yksilöi resurssin.

1 Johdanto

Bittium Oyj on suomalainen tuote- sekä ohjelmistoyritys, jonka tuotteisiin kuuluvat erilaiset turvalliset viestintä- sekä liitettävyysratkaisut. Opinnäytetyö tehtiin toimeksiantona Bittium Wireless Oy:lle, joka on osa Bittium Oyj:tä ja vastaa yhtiön tämänhetkisen päätuotteen, Bittium **TAC WIN** -verkkoympäristön kehittämisestä. Järjestelmään kuuluu taktisia reitittimiä, radioita sekä päätelaitteita, joiden avulla voidaan muodostaa nopeasti **MANET**-verkkoja.

Bittium TAC WIN -ympäristö mahdollistaa itsenäisen ja laajakaistaisen IP-verkon muodostamisen. Verkko voidaan muodostaa paikasta riippumatta käyttäen TAC WIN –tuoteperheen radioita, antenneja, reitittimiä sekä päätelaitteita. Verkon yli voidaan välittää tietoa, kuvaa sekä ääntä. Verkko on dynaaminen sekä itseään korjaava ja sen yli siirrettävä tieto voidaan salata. TAC WIN -ympäristö tarjoaa myös rajapintoja, joiden kautta ulkopuoliset laitteet voivat tarvittaessa/sallittaessa liittyä verkkoon. Ympäristöön sekä sitä käyttäviin tuotteisiin on mahdollista integroida asiakkaan käyttämiä laitteita sekä palveluita [1].

IP-verkossa käytettävien laitteiden konfigurointi on oleellinen osa verkon muodostusta ja ylläpidettävyyttä. Verkon muodostamiseen tarvittavien parametrien määrä sekä laitteiden tarjoamien palveluiden tarvitsemien parametrien määrä on kasvanut verkkoympäristön kehitystyön mukana. Nykyisellään konfiguraatiotiedostot voivat olla useita satoja rivejä pitkiä ja niiden muokkaaminen tapahtuu muokkaamalla suoraan laitteiden ymmärtämiä tiedostomuotoja, jotka eivät ole luettavuudeltaan ihmiselle optimaalisia. Lisäksi konfiguraatiotiedoston tekijällä täytyy olla vahva ymmärrys muokattavien parametrien sallituista arvoista sekä rajoista. Nämä seikat ovat nostaneet esiin tarpeen työkalulle, jolla konfiguraatioiden luominen, muokkaaminen ja varmentaminen olisi selkeämpää, käyttäjäystävällisempää sekä nopeampaa.

Opinnäytetyön tarkoituksena oli suunnitella, toteuttaa sekä dokumentoida tällainen työkalu sen ensimmäiseen vaiheeseen. Työkalun toteutuksen ensimmäiselle vaiheelle oli määritelty työnohjaajan toimesta vaatimukset, jotka sen tulisi täyttää. Vaikka työkalu suunniteltiin ja toteutettiin TACWIN-ympäristö edellä, pyrittiin suunnittelu- ja toteutusvaiheessa mahdollisimman geneerisiin ja joustaviin ratkaisuihin. Tällä tähdättiin helppoon jatkokehitettävyyteen sekä muokattavuuteen. Yrityksessä on aiemmin toteutettu demo-ohjelmisto, jonka tarkoituksena oli osoittaa idean toimivuus. Demo-ohjelmiston kehitys on lopetettu eikä se ole käytössä. Opinnäytetyön teon yhteydessä perehdyttiin demo-ohjelmistoon ja todettiin sen perusidea toimivaksi, mutta toteutus ja käyttö sekavaksi.

TACWIN-tuotteiden konfiguraatioissa käytetään suurelta osin JSON-formaattia olevia tiedostoja. Lisäksi osaan tuotteista on integroitu ohjelmistollinen sisäinen varmennus, joka hyödyntää **JSON Schema** –standardin implementoivia ohjelmia. Tällä sisäisellä varmennuksella pyritään varmistamaan, ettei käyttäjä voi konfiguroida virheellisiä arvoja. Opinnäytetyössä syntynyt ohjelmisto pyrki tuomaan tällaisen varmennuksen jo itse konfiguraatiotiedoston tekovaiheeseen, jolloin verkon operaattorit voivat välttää virheelliset arvot jo konfiguraation tekovaiheessa.

2 Työkalun suunnittelu ja dokumentointi

Työkalun toteutuksessa otettiin lähtökohdaksi JSON-muotoisten konfiguraatiotiedostojen luonti sekä validointi JSON Scheman mukaisia malleja vasten. Malleilla voidaan rajata ja varmentaa JSON-tiedoston sisältöä. XML-formaatti jätettiin pois työkalun vaatimuksista, koska se on jäämässä hiljalleen pois konfigurointiformaattina eikä sen validointia tueta laitteissa. XML-formaatti on ihmiselle huomattavasti vaikealukuisempaa verrattuna JSON-formaattiin.

Koska ohjelman tuli olla helposti käyttöönotettavissa sekä toimia eri alustoilla, päädyttiin se toteuttamaan verkkoselainta alustana käyttäen.

Ohjelmiston lähdekoodin sekä dokumenttien versionhallinta toteutettiin Git:llä.

2.1 Perusvaatimukset

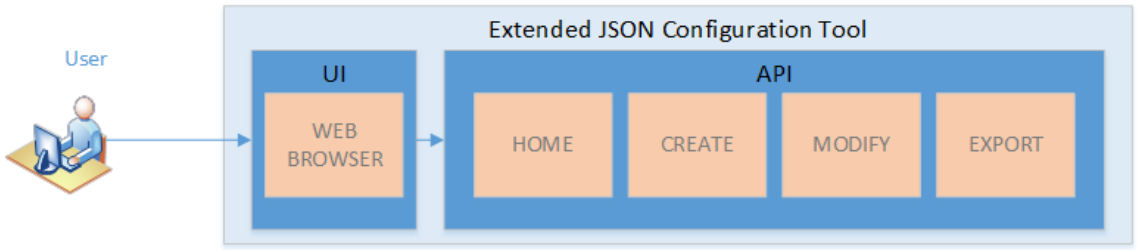
Työkalulle asetettiin tilaajan toimesta kuusi perusvaatimusta, joiden tuli täyttyä työkalun ohjelman implementaation ensimmäisessä vaiheessa:

- mahdollisuus tuoda JSON-formaatissa olevia tiedostoja työkaluun muokattavaksi
- mahdollisuus luoda konfiguraatioita JSON Scheman vaatimukset täyttävien mallien pohjalta, käyttäen mallien linkitystä
- mahdollisuus muokata konfiguraation yksittäisiä parametrejä
- mahdollisuus varmentaa sekä koko konfiguraatio että sen osakomponentit malleja vasten
- mahdollisuus tallentaa muokattu ja varmennettu konfiguraatio käyttäjän tiedostojärjestelmään
- käyttökelpoinen Windows- sekä Linux-käyttöjärjestelmillä PC-ympäristössä

Lisäksi työkalun tuli olla dokumentoitu sekä jatkokehitettävä. Myös mahdollinen integrointi muihin järjestelmiin tuli ottaa huomioon.

2.2 Ohjelmistoarkkitehtuuri

Ohjelmisto pyrittiin suunnittelemaan mahdollisimman modulaariseksi. Suunnittelussa käyttöliittymä eriytettiin ohjelmiston liiketoiminnallisen kerroksesta, joka niin ikään jaettiin eri osakomponentteihin. Ohjelmiston liiketoiminnallinen kerros päätettiin paljastaa käyttöliittymälle ohjelmointirajapinnan (**API**) kautta. Ohjelmiston perusvaatimusten pohjalta muodostui kolme laajempaa kokonaisuutta, joiden pohjalta liiketoimintakerrosta alettiin jäsentämään.



Kuva 1. Yksinkertaistus suunnitellusta ohjelmistoarkkitehtuurista

2.2.1 Käyttöliittymäarkkitehtuuri

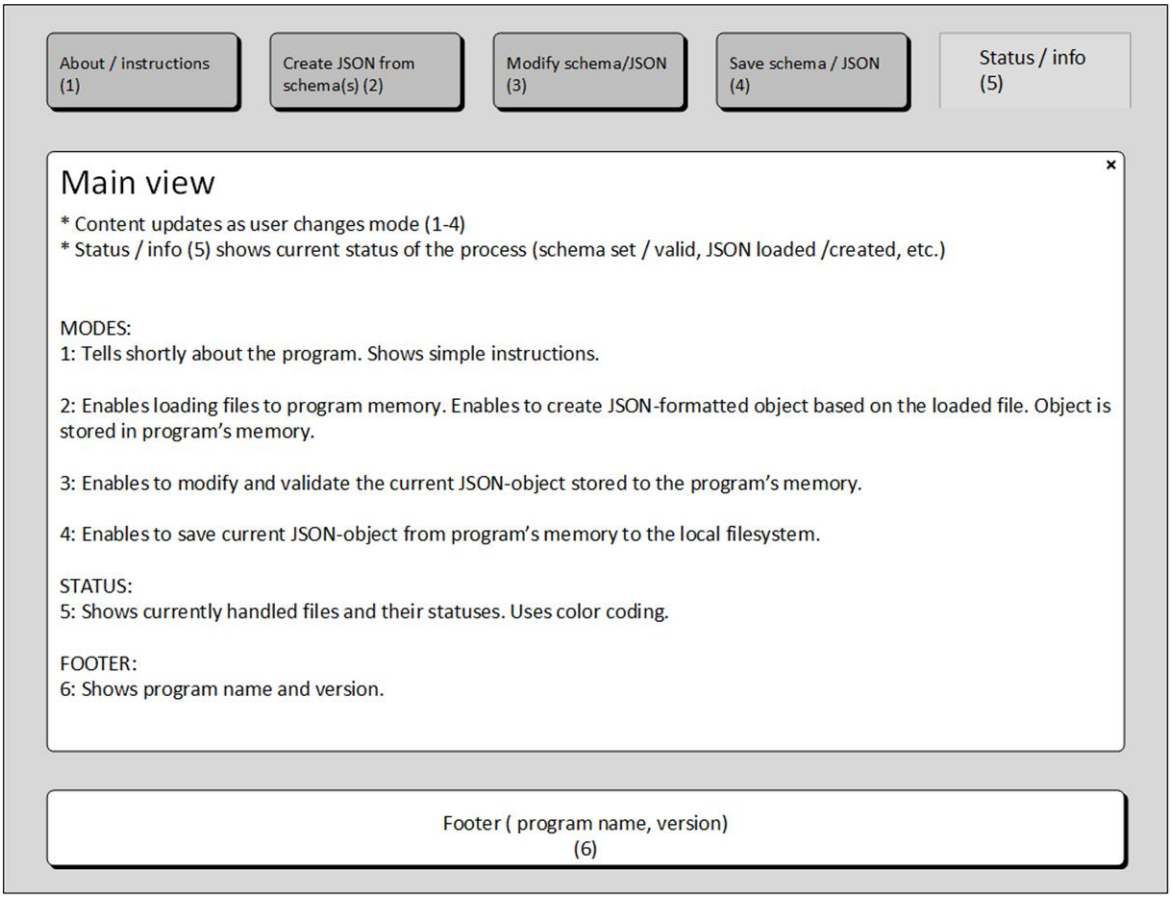
Käyttöliittymää suunnitellessa otettiin huomioon ensimmäisen vaiheen toteutuksen verrattainen yksinkertaisuus käyttöliittymän osalta. Käyttöliittymä päätettiin jakaa eri osioihin käyttäen apuna aiemmin jäsenneltyjä liiketoimintakerroksen osakomponentteja. Osiot suunniteltiin jakamaan käyttöliittymän tarjoama tila siten, että jokainen osio muodostaa verkkoselaimella oman itsenäisen sivun, joiden välillä pystyy vaihtamaan käyttöliittymään ylälaitaan sidotuilla napeilla. Sivujen lähtökohtaiset funktionaalisuudet määriteltiin seuraavasti:

- sivu, jolla on mahdollista joko luoda uusi konfiguraatio tai ladata jo olemassa oleva
- sivu, jolla on mahdollista muokata konfiguraatiota sekä varmentaa se
- sivu, jolla on mahdollista tallentaa konfiguraatio käyttäjän tiedostojärjestelmään

Lisäksi päätettiin luoda niin kutsuttu 'kotisivu', joka sisältäisi lyhyen esittelyn työkalusta.

Koska ohjelmisto ei käytä varsinaista serveriä eikä käyttöliittymän vaatiman koodin määrä vaikuttanut suurelta, päätettiin käyttöliittymä toteuttaa niin kutsutulla **SPA**-mallilla. Tämä mahdollisti käyttöliittymän HTML-kuvauksen sijoittamisen yhteen tiedostoon, joka vuorostaan

helpotti erilaisia kokeiluja käyttöliittymän käytön suhteen. Käyttöliittymästä laadittiin muutamia alustavia luonnoksia. Käyttöliittymän värimaailmaa ei suunniteltu vielä tässä vaiheessa.



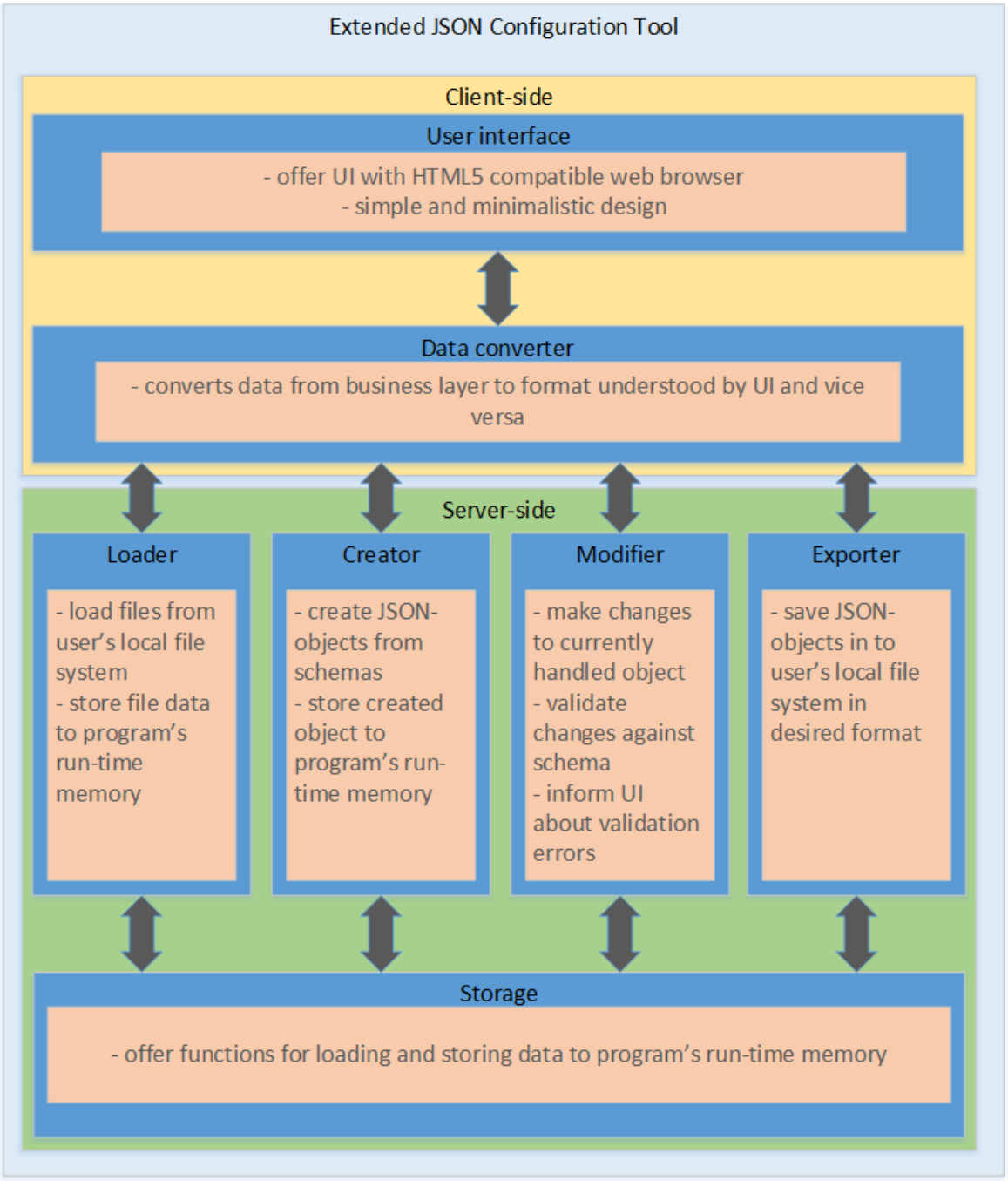
Kuva 2. Suunnitellun käyttöliittymän varhaista luonnostelua

2.2.2 Liiketoimintakerroksen arkkitehtuuri

Liiketoimintakerroksen arkkitehtuuria suunnitellessa otettiin huomioon uudelleenkäytettävyys sekä mahdollinen integroitavuus muihin tuotteisiin. Ajatuksena oli, että liiketoimintakerros on kokonaan riippumaton käyttöliittymästä. Tällainen suunnittelu mahdollistaa sen, että liiketoimintakerros voidaan siirtää sellaiseen toiseen tuotteeseen tai käyttöliittymä voidaan kirjoittaa uusiksi ilman, että liiketoimintakerroksen toiminnallisuuteen tarvitsee tehdä muutoksia.

Koska perusvaatimusten pohjalta laaditut laajemmat kokonaisuudet vaikuttivat selkeiltä osilta, päätettiin liiketoimintakerroksen arkkitehtuuri jakaa siten, että ohjelmointirajapinta tarjoaisi neljä eri luokkaa, joiden sisältämillä funktioilla käyttöliittymä toteuttaisi tarvittavat tehtävät.

Lisäksi arkkitehtuurin pohjatasolle sijoitettiin yleishyödylliset funktiot sekä funktiot, jotka käsittelevät web-selaimen tarjoamaa väliaikaista tallennustilaa.



Kuva 3. Kuvaus ohjelmiston suunnitellusta arkkitehtuurista

2.3 Avoimen lähdekoodin kirjastot

JSON Schemaa noudattavien mallien käyttöön löytyy varsin kattava valikoima kirjastoja, joista suurin osa on julkaistu avoimen lähdekoodin lisenssillä. Käytettävien kirjastojen tuli kuitenkin sallia lähdekoodin vapaa muokkaus sekä yrityksen kirjastoa käyttävän tuotteen vapaa julkaisu ilman yrityksen lähdekoodin julkistamista. Näin ollen sallitut vaihtoehdot rajoittuivat BSD-tyyppisiin, vapaan muokkaamisen ja julkaisun salliviin lisensseihin. Tällaiset lisenssit sallivat teoksen käytön, muokkaamisen ja julkaisun osana yrityksen tuotetta, kunhan ohjelman mukana toimitetaan teoksen tekijöiden nimet sekä viittaus alkuperäiseen lisenssiin. [2.] [3.]

Ensimmäisenä kartoitettiin JSON Scheman implementoivia kirjastoja. Yrityksessä aiemmin toteutettu demo käytti validointiin *JSON Editor* –nimistä, MIT-lisenssiä käyttävää kirjastoa. Lyhyen kartoituksen sekä pienen demon teon jälkeen ohjelmistossa päädyttiin käyttämään kyseistä kirjastoa.

JSON Editor tarjosi käyttökelpoisen lisensoinnin, mahdollisuuden luoda JSON-tiedostoja mallien pohjalta sekä sitä kehitettiin versionhallinnan perusteella aktiivisesti. Lisäksi se oli kirjoitettu siten, ettei sillä ollut riippuvuuksia: kirjastoa pystyi käyttämään sisällyttämällä yhden tiedoston omaan ohjelmistoon. [4.]

Käyttöliittymän toteuttavaan verkkosivuun kirjastot haettiin samankaltaisilla perusteilla. Aluksi verkkosivua lähdettiin kehittämään *jQuery*-nimistä Javascript-kirjastoa apuna käyttäen [5]. Vaikka *jQuery* on saavuttanut ajansaatossa suurta suosiota, huomattiin pian käyttöliittymän implementoinnin aloittamisen jälkeen, että kirjastolla on tarpeettoman kömpelö toteuttaa reaktiivisia sovelluksia. Koska ohjelmiston täytyi jatkuvasti pystyä aktiivisesti reagoimaan käyttäjän syötteisiin, ja koska kehitystyön alkuvaiheessa käyttöliittymän kehitys oli iteratiivista, päädyttiin rinnalle ottamaan *Vue.js*-kirjasto [6]. *Vue.js* tarjosi *jQuery*ä helpomman sekä nopeamman tavan toteuttaa käyttöliittymäelementtejä. *jQuery* pidettiin mukana, koska se sisälsi muutamia toteutusta nopeuttavia funktionaalisuuksia, kuten asynkronisten pyyntöjen lähettämisen. Kirjastojen rinnakkaisessa käytössä täytyi ottaa huomioon mahdolliset kirjastojen väliset konfliktit.

3 JSON-formaatti

JSON on kevyt, selkeälukuinen sekä kohtuu helposti tulkittava tiedonvaihtoformaatti. Sen juuret ovat Javascript-ohjelmointikielessä, mutta se noudattaa yleisesti ottaen kaikissa ohjelmointikielissä tuettua rakennetta, jossa tieto koostuu objekteista, listoista sekä näiden sisältämistä ominaisuuksista. Ominaisuuksia kutsutaan myös avain-arvo-pareiksi [7]. Objekti on itsessään avain-arvo-pari. Avaimia kuvataan lainausmerkkien sisään sijoitetulla merkkijonolla. Lista sisältää arvoja säilyttäen niiden järjestyksen. Arvot ovat ohjelmointikielille tyypillisiä tietotyyppejä [8]:

- *object*
    - objekti, merkitään aaltosulkeilla, esimerkiksi *"key": { "value": 1 }*
    - voi olla tyhjä tai sisältää *n*-määrän muita tietotyyppejä
- *array*
    - lista, merkitään hakasulkeilla, esimerkiksi *"locklist": [1, 2, 3]*
    - voi olla tyhjä tai sisältää *n*-määrän muita tietotyyppejä
- *number*
    - kokonais- tai desimaaliluku, esimerkiksi *"integer": 1* tai *"decimal": 3.14*
- *string*
    - merkkijono, merkitään lainausmerkeillä *""*, esimerkiksi *"merkkijono": "arvo"*
- *boolean*
    - totuusarvo, merkitään *true* tai *false*, esimerkiksi *"thesis_done": true*
- *null*
    - "nolla", olematon, merkitään *null*, esimerkiksi *"yet_non_existing: null"*
    - käytetään yleisesti merkitsemään puuttuvaa ja/tai alustamatonta arvoa

JSON-tiedosto tai tietorakenne kirjoitetaan siten, että nimettömän objektin sisään sijoitetaan avain-arvo-pareja. Näiden parien arvot voivat olla joko puhtaita arvoja (*string*, *number*, *boolean*, *null*) tai objekteja tai listoja. JSON-syntaksi ei salli kommentointia.

Yksinkertainen, henkilötietoja kuvaava JSON-objekti voitaisiin kirjoittaa seuraavalla tavalla:

```
1. {
2.   "name": "John Doe",
3.   "birthday": {
4.     "day": 23,
5.     "month": "January"
6.   },
7.   "thesis_done": true,
8.   "email": [
9.     "some@place.com",
10.      "other@service.com"
11.    ]
12.  }
```

Jos jokin palvelu käyttäisi tällaista pohjaa henkilötietojen tallennukseen, tulisi sen tarkistaa, että avainten arvot ovat järkevissä rajoissa: ikä ei voi olla negatiivinen, syntymäpäivän tulee olla välillä 1-31 hieman kuukaudesta riippuen, sähköpostiosoitteen tulee tietyn muotoinen ja niin edelleen. Tarkistukset voidaan kirjoittaa suoraan koodiin, mutta tällainen lähestymistapa on hankala ylläpidettävä sekä hidas muokata. Käyttämällä JSON Scheman mukaista mallia syötteen vertailupohjana, voidaan tarkistamisesta tehdä geneeristä sekä helposti ylläpidettävää.

4 JSON Schema

JSON Schema on JSON-formaattia "noudattava sanasto, joka mahdollistaa JSON-dokumenttien validoinnin, kommentoinnin sekä muokkaamisen" [9]. JSON Schemaa kehitetään ja ylläpidetään vapaaehtoisvoimin. JSON Scheman avulla voidaan varmistaa, että JSON-formaattia oleva tiedosto noudattaa rakenteellisesti haluttua kaavaa. Jatkossa dokumentissa puhutaan JSON Schemasta, kun halutaan viitata standardointia kehittävään tahoon ja kun puhutaan JSON-malleista, tarkoitetaan JSON Schema -standardia noudattavia JSON-tiedostoja, joita on tarkoitus käyttää tietojen validointiin.

JSON-mallit ovat itsessään JSON-formaattia noudattavia tiedostoja tai tietorakenteita, jotka voivat koostua sisäisesti useista osamalleista. Tiedostoille ei ole määritelty erillistä tiedostopäätettä, mutta niiden **MIME**-tyyppi on *application/schema+json* [10]. Yleinen käytäntö on käyttää joko *.json*- tai *.schema.json* -tiedostopäätettä.

JSON Schemasta pyritään pitämään yllä neljää Internet-luonnosta (**Internet-Draft**). Luonnokset käsittelevät JSON-mallien eri ominaisuuksia sekä yleistä rakennetta. Pääluonnos on nimeltään *draft-handrews-json-schema-01*. Ylläpitäjät pyrkivät luonnosten säännölliseen päivittämiseen. Lähteen hakuhetkellä viimeisin ydinosaa koskeva luonnos oli määrätty päättymään syyskuun 20. päivä 2018. [11.]

Sisäisesti JSON Schema käyttää versionumerointityyliä *draft-xx*, jossa *xx* on versionumero. Kirjoitushetkellä viimeisin virallisesti julkaistu versio on *draft-07*. Merkittävät versiot ovat *draft-04*, *draft-06* sekä *draft-07*.

4.1 Mallien perusteet

JSON Schema määrittelee joukon avainsanoja, joita käytetään mallin luomisessa. Näiden avainsanojen avulla mallit implementoiva ohjelma pystyy tarkistamaan avain-arvo-parien oikeellisuuden sekä kuvaamaan niiden käyttöä. Lisäksi avainsanoilla on mahdollista linkittää JSON-malleja toisiinsa, jolloin mallit voidaan jakaa pienempiin osiin, tarvittaessa eri tiedostoihin ja siten helpottaa uudelleenkäyttöä sekä muokattavuutta.

Avainsanoja on lukuisia ja osaan on tullut muutoksia uusien versioiden myötä. On huomattava, että koska malli on itsessään JSON-tiedosto tai tietorakenne, tulee mallien avainsanoja välttää itse käsiteltävää tietoa luotaessa. Jäljempänä mainitut avainsanat ja niiden kuvaukset on koostettu version *draft-07* mukaan [12].

4.2 Merkintä sekä annotaatiot

Koska JSON-mallit ovat myös JSON-objekteja, on niillä avainsanat, joilla ne voidaan tunnistaa sekä yksilöidä. Lisäksi ne voivat kertoa, minkä version mukaan malli on kirjoitettu. Tunnisteet on hyvä sijoittaa mallin alkuun.

Avainsanaa *$schema* käytetään tiedottamaan, että kyseinen tiedosto on JSON-malli:

```
1. {"$schema": "extended-json-conf-tool/draft-07/schema#"}
```

Yllä oleva julistus kertoo myös käytetyn version sisällyttämällä sen merkkijonoon. [13.]

Avainsanalla *$id* voidaan yksilöidä malli:

```
1. {"$id": "extended-json-conf-tool/schemas/mainschema.json"}
```

$id-avainsanaa voidaan käyttää myös mallien linkityksessä (*4.5 Mallien linkittäminen ja yhdistäminen*). [14.]

Mallit sallivat myös käsiteltävän tiedon annotoinnin. Siten implementoiva ohjelma pystyy näyttämään loppukäyttäjälle hyödyllistä tietoa avain-arvo-pareihin liittyen. Nämä annotointiin liittyvät avainsanat ovat:

- *title*
    - Otsikkotason kuvaus annettavasta tiedosta
- *description*
    - Yksityiskohtaisempi kuvaus annettavasta tiedosta ja sen tarkoituksesta
- *default*
    - Vakio, jolla arvo alustetaan
- *examples*
    - Lista esimerkkiarvoista

Avainsana *$comment* on tarkoitettu mallin ylläpitäjien avuksi. Avainsanan tarkoituksena on mahdollistaa itse mallin kommentointi ilman, että se vaikuttaa loppukäyttäjään millään tavalla. [15.]

Asiakastunnistetta kuvaava malli, jossa käytetään annotaatioita sekä kommentointia, voisi olla alla olevan kaltainen.

```
1. {
2.   "title": "Client ID",
3.   "description": "Client ID as 16 character string",
4.   "type": "string",
5.   "default": "abcdeFghijKlMnop",
6.   "examples": [
7.     "jaiEpqNxzanifuei",
8.     "iaeSQnxwefaIpqDd"
9.   ],
10.    "$comment": "TODO: better example values"
11.  }
```

JSON-mallit implementoiva ohjelma pystyisi mallin avulla tarjoamaan tietoa käyttäjälle siitä, millainen asiakastunnisteen tulee olla. Kommentoinnin avulla mallin ylläpito helpottuu, kun parannuskohteet ja huomiot voidaan kirjoittaa suoraan ylläpidettävään tiedostoon.

Mitkään merkintään ja annotaatioon liittyvät avainsanat eivät ole pakollisia mallien määrittelyssä.

4.3 *type*- ja *enum*-avainsanat

JSON-mallien *type*-avainsanalla voidaan määrittää, mitä tietotyyppejä JSON-tiedoston jonkin avaimen arvo voi ilmentää [16]. Tyypit, jotka avainsanalla voidaan määrittää, noudattavat pääosin JSON-formaatin tyyppejä. Poikkeuksen muodostavat numeeriset tyypit: JSON-formaatti määrittelee numeeriset arvot *number*-tyyppisiksi, eikä ota kantaa siihen, onko arvo kokonais- vai desimaaliluku. JSON-malleissa pystytään kuitenkin määrittämään voiko numeerinen arvo olla desimaali- tai kokonaisluku (*number*) vai pelkästään kokonaisluku (*integer*).

type-avainsanan arvo kirjoitetaan joko merkkijonona tai listana merkkijonoja:

```
1. {"type": "integer"}
2. {"type": ["array", "object"]}
```

enum-avainsanalla voidaan määrittää lista arvoja, jotka avain voi saada:

```
1. "type": "integer",
2. "enum": [1, 22, 333, 4444]
```

Mallin mukainen, kokonaislukutyyppiä oleva avain voi saada arvon *enum*-avainsanan määrittämästä joukosta: 1, 22, 333 tai 4444. [17.]

4.4 Tyypit sekä tyyppikohtaiset avainsanat

JSON-mallien tyypit käyttävät eri ominaisuuksiensa johdosta eri avainsanoja kuvaamaan niiden rakennetta. Seuraavat esimerkit antavat yksinkertaisen kuvauksen eri tyyppien avainsanoista sekä niiden käytöstä. Silloin kun tieto tyypin avainsanojen käyttöön on haettu yhdestä lähteestä, on viittaus lähteeseen sijoitettu tyypin kattavan otsikon viimeisen kappaleen jälkeen. Tyyppeihin suoraan liittymättömät viittaukset on sijoitettu normaalisti.

4.4.1 object

object-tyyppiä käytetään avaimen arvona, kun halutaan luoda sisäkkäisiä rakenteita. Koska ainoastaan *object*-tyyppi yhdessä *array*-tyypin kanssa pystyy sisältämään mitä vain muita tyyppejä, luovat ne yhdessä JSON-tiedostojen puurakenteen:

```
1. {
2.   "object1": {
3.     "array1": [
4.       {
5.         "object2": {
6.           "object3": 100,
7.           "array2": [
8.             "value1"
9.           ]
10.         }
11.       },
12.       "value2"
13.     ]
14.   },
15.   "object4": {}
16. }
```

Kun dokumentissa jatkossa puhutaan objektista, tarkoitetaan avain-arvo-paria, jonka arvo on tyyppiä *object*.

Perusominaisuudet

object-tyypin mahdollisia arvoja voidaan määrittää ja rajata useilla eri avainsanoilla:

- *properties*-avainsanaa käytetään kuvaamaan sitä, millaisia avain-arvo-pareja objekti sisältää
- *required*-avainsanan arvo on lista merkkijonoja, joilla kerrotaan ne *properties*-avainsanalla kuvatut avain-arvo-parit, joiden täytyy olla mukana luodussa objektissa
- *additionalProperties*-avainsanan arvo on joko totuusarvo, joka määrittää saako objektissa olla muita kuin *properties*-avainsanalla kuvattuja avain-arvo-pareja tai sitten objekti, jonka sisään kirjoitetaan mitä tyyppiä lisättävät objektit voivat olla

```
1. {
2.   "car": {
3.     "type": "object",
4.     "properties": {
5.       "tires": "integer",
6.       "color": "string",
7.       "roof": "boolean"
8.     },
9.     "required": [
10.        "tires", "color"
11.     ],
12.     "additionalProperties": false
13.   }
14. }
```

Yllä kuvatun mallin mukaisessa *car*-nimisessä objektissa tulee olla mukana *tires*- ja *color*-avaimet oikeilla arvotyypeillä. Lisäksi siinä voi olla mukana *roof*-avain, jonka arvo on *boolean*-tyyppiä, mutta muiden avainten luonti on estetty asettamalla *additionalProperties*-avaimen arvo epätodeksi.

Jos ylläolevan mallin haluttaisiin sallivan ylimääräiset avaimet vain, jos niiden arvot olisivat *string*-tyyppiä, kirjoitettaisiin *additionalProperties* seuraavasti:

```
1.   "additionalProperties": {
2.     "type": "string"
3.   }
```

min- ja maxProperties

Objektin sisältämien avainten määrää voidaan rajoittaa *minProperties* ja *maxProperties*-avainsanoilla, joista ensimmäinen asettaa alarajan ja jälkimmäinen ylärajan. Avainsanojen arvo tulee antaa kokonaislukuna ilman lainausmerkkejä.

```
1. {"minProperties": 7}
```

propertyNames

Jos halutaan asettaa rajoituksia sille, minkä nimisiä avaimia objektiin voidaan lisätä, voidaan käyttää *propertyNames*-avainsanaa:

```
1. {
2.   "propertyNames": {
3.     "pattern": "^(key[0-9])$"
4.   }
5. }
```

Ylläolevaa mallia mukaileva objekti voi sisältää avaimia, jos niiden nimi sisältää alussa sanan *key* ja sen jälkeen numeron väliltä 0-9, esimerkiksi *key5* tai *key7master*.

dependencies

Objektin sisältämien avainten välille voidaan luoda riippuvuussuhteita *dependencies*-avainsanan avulla. Avainsanan saama arvo on itsessään objekti, johon riippuvuussuhteet määritetään objektien avaimia käyttäen. Alla oleva esimerkkimalli kuvaa reaalinumeroa, jolla on kantaluku, mutta eksponentti ei ole pakollinen. Toisaalta eksponentti ei voi esiintyä yksinään, koska sillä on riippuvuussuhde kantalukuun.

```
1. {
2.   "real_number": {
3.     "type": "object",
4.     "properties": {
5.       "base": {
6.         "type": "number"
7.       },
8.       "exponent": {
9.         "type": "number"
10.        }
11.      }
12.    },
13.    "additionalProperties": false,
14.    "dependencies": {
15.      "exponent": ["base"]
16.    }
17.  }
```

patternProperties

Avainsanan avulla on mahdollista määrittää, millaisia merkkijonorakenteita avaimien nimet voivat saada sekä millaisia arvoja avaimet voivat nimensä perusteella saada:

```
1. {
2.   "type": "object",
3.   "additionalProperties": false,
4.   "patternProperties": {
5.     "^(key[0-9])$": {
6.       "type": "integer"
7.     },
8.     "^(lock[A-J])$": {
9.       "type": "string",
10.        "minLength": 4
11.      }
12.    }
13.  }
```

Säännöllisten lausekkeiden määrittämät mahdolliset avaimet ovat joko *keyX* tai *lockZ*, jossa *X* on numero väliltä 0-9 ja *Z* on iso kirjain väliltä A-J. Näiden avainten arvojen tyypit ovat vastaavasti joko kokonaisluku tai merkkijono. Merkkijonolle on lisävaatimuksena neljän merkin vähimmäispituus.

[7.]

4.4.2 array

array-tyyppi kuvaa avaimen arvoa lähtökohtaisesti listana. Useat ohjelmointikielet tuntevat listan lisäksi myös monikon (engl. *tuple*). Tietojenkäsittelyssä monikolla tarkoitetaan listaa, jonka arvot ovat muuttumattomia [18]. Lähtökohtaisesti JSON-formaatin listat voivat sisältää *n*-määrän mitä vain tyyppejä. Elementtejä pystytään lisäämään ja poistamaan. Mallien avulla listan sisältö voidaan määrätä listan pituuden ja sen sisältämien tyyppien mukaan, jolloin *array*-tyypillä pystytään kuvaamaan myös monikkoja.

Perusominaisuudet

Listan pituudelle, eli sen sisältämien elementtien määrälle, voidaan asettaa ehtoja. Avainsanoilla *minItems* ja *maxItems* määrätään elementtien minimi- sekä maksimimäärä.

uniqueItems-avainsanalla voidaan kertoa, että listan sisältämien elementtien tulee olla yksilöllisiä. Avainsanan saama arvo on totuusarvo.

```
1. {
2.   "type": "array",
3.   "uniqueItems": true,
4.   "minItems": 5,
5.   "maxItems": 10
6. }
```

Ylläolevan mallin mukaisessa listassa tulee olla vähintään viisi ja enintään kymmenen, yksilöllistä arvoa. Arvot voivat mitä vain tyyppejä.

contains

contains-avainsanalla voidaan kertoa, millaisia tyyppejä listan tulee sisältää. Alle kuvatun mallin mukaisen listan tulee sisältää ainakin yksi kokonaisluku- sekä yksi merkkijonotyyppinen elementti.

```
1. {
2.   "type": "array",
3.   "contains": {
4.     "type": "integer",
5.     "type": "string"
6.   }
7. }
```

On huomattava, että *contains*-avainsana ei ota kantaa siihen, mitä muita tyyppejä lista sisältää: ylläolevan mallin mukainen lista voi sisältää kokonaisluvun sekä merkkijonon lisäksi *n*-määrän muita tyyppejä.

items ja *additionalItems*

items-avainsanaa voidaan käyttää kertomaan, millaisia tyyppejä lista sisältää. Jos halutaan kuvata listaa, voidaan avainsanan arvona käyttää objektia, jonka sisälle on määritelty listassa sallittavat tyypit. Yksinkertainen, vain kokonaislukuja sisältävä lista voidaan kuvata seuraavalla mallilla:

```
1. {
2.   "type": "array",
3.   "items": {
4.     "type": "integer"
5.   }
6. }
```

Jos halutaan kuvata monikkoa, voidaan avainsanan arvo antaa listana, joka sisältää halutut tyypit halutussa järjestyksessä.

```
1. {
2.   "type": "array",
3.   "items": [
4.     {
5.       "type": "integer"
6.     },
7.     {
8.       "type": "string"
9.     },
10.      {
11.        "type": "object",
12.        "properties": {
13.          "type": "number"
14.        }
15.      }
16.    ]
17.  }
```

Ylläolevan mallin mukaisen monikon tulee sisältää, annettua järjestystä noudattaen, kokonaisluku, merkkijono sekä mallin mukainen objekti.

Jos monikkoa kuvaavan mallin täytyy sallia, estää tai kuvata ylimääräisiä arvoja, voidaan tämä toteuttaa *additionalItems*-avainsanalla. Avainsana saa arvokseen joko totuusarvon tai objektin, jossa kuvataan sallitut ylimääräiset arvot. Esimerkki, joka sallii ylimääräiset arvot tyypistä riippumatta:

```
1. {
2.   "type": "array",
3.   "items": [
4.     {"type": "integer"},
5.     {"type": "string"},
6.     {"type": "object"}
7.   ],
8.   "additionalItems": true
9. }
```

Esimerkki, joka sallii ylimääräiset arvot vain, jos ne ovat neljän merkin mittaisia merkkijonoja:

```
1. {
2.   "type": "array",
3.   "items": [
4.     {"type": "integer"},
5.     {"type": "string"},
6.     {"type": "object"}
7.   ],
8.   "additionalItems": {
9.     "type": "string",
10.      "minLength": 4,
11.      "maxLength": 4
12.    }
13.  }
```

[19.]

4.4.3 string

string-tyyppi kuvaa avaimen arvoa merkkijonona. Merkkijono kirjoitetaan lainausmerkkien (”“) väliin. JSON-mallien avulla voidaan määrittää merkkijonon pituus sekä haluttu rakenne. Merkkijonon pituus kerrotaan *minLength* ja *maxLength* avainsanoilla. Säännöllisillä lausekkeilla (**regex**) voidaan määrätä merkkijonon haluttu rakenne. Säännöllinen lauseke merkitään *pattern*-avaimella.

Esimerkkimalli, jossa määritellään rakenne lyhyelle tekstille, joka saa sisältää vain kirjaimia väliltä a-z joko pienenä tai isona kirjaimena:

```
1. {
2.   "type": "string",
3.   "minLength": 0,
4.   "maxLength": 64,
5.   "pattern": "^[a-zA-Z]$"
6. }
```

JSON Schema määrittelee *string*-tyypille myös avainsanan *format*, jota voidaan käyttää merkkijonon rakenteen määrittelyssä. Valmiita, JSON Schemaan kuuluvia formaatteja on lukuisia, mutta on huomattava, että ollakseen JSON Schema –yhteensopiva, ohjelman ei tarvitse implementoida *format*-avainsanaa. Seuraava esimerkki käyttää arvoa *date*, jonka myötä *string*-tyypin arvon tulee olla muodoltaan *yyyy-mm-dd*-formaattia, esimerkiksi *"2019-01-23"*.

```
1. {
2.   "type": "string",
3.   "format": "date"
4. }
```

[20.]

4.4.4 *number* ja *integer*

Numeerisille tyypeille, *number* ja *integer*, voidaan asettaa ala- ja ylärajat. Rajat asetetaan *minimum*, *maximum*, *exclusiveMinimum* ja *exclusiveMaximum* avainsanoilla. Ensimmäiset kaksi avainsanaa määrittävät lukujoukon siten, että annettu arvo sisältyy sallittuun lukujoukkoon. Jälkimmäiset kaksi eivät sisällytä annettua arvoa sallittuun lukujoukkoon Lisäksi arvoille voidaan asettaa vaatimus siitä, että ne ovat jonkin annetun luvun kerrannaisia. Kerrannaisuusvaatimus

asetetaan *multipleOf*-avainsanalla. Arvot annetaan numeroina eivätkä ne saa olla *string*-muotoisia: arvo *"42"* ei ole numeerinen tyyppi.

Esimerkki numeeristen tyyppien käytöstä mallissa, joka sallii avaimelle *decade* arvot välillä 0-90, voisi olla seuraavan kaltainen:

```
1. {
2.   "decade": {
3.     "type": "integer",
4.     "minimum": 0,
5.     "exclusiveMaximum": 100,
6.     "multipleOf": 10
7.   }
8. }
```

[21.]

4.4.5 boolean

Totuusarvoa merkitään *boolean*-tyypillä. Sen arvo voi olla joko tosi (*true*) tai epätosi (*false*). Arvot kirjoitetaan ilman lainausmerkkejä. On huomattava, että arvot, jotka ohjelmointikielissä pystytään yleensä käsittelemään totuusarvoina, eivät ole sallittuja arvoja JSON-malleissa. Siksi esimerkiksi numeerisella *0*:lla ei pystytä merkitsemään arvoa epätotena.

Esimerkki totuusarvon käytöstä mallissa, jossa *enabled*-avain saa vakiona arvon *true*:

```
1. {
2.   "enabled": {
3.     "type": "boolean",
4.     "default": true
5.   }
6. }
```

[22.]

4.4.6 null

Yleisesti tyyppiä *null* käytetään merkitsemään puuttuvaa arvoa. JSON-mallissa esimerkki sen käytöstä voisi olla tilanne, jossa jokin avain on olemassa koko ajan, mutta avaimen tyyppi ja arvo halutaan määrittää ohjelmassa vasta ajonaikaisesti tai manuaalisesti jälkikäteen.

```
{
  "client": {
    "required": [
      "name", "additionalData"
    ],
    "properties": {
      "name": {
       "type": "string"
      }
        "additionalData": {
          "type": null
         }
      }
    }
  }
```

Yllä olevan mallin mukainen JSON-tiedosto sisältää *additionalData*-avaimen, mutta malli ei ota kantaa sen tulevaan tyyppiin tai arvoon. On huomattava, että *null*-tyyppinen avain ei validointihetkellä voi saada muuta arvoa kuin *null*.

[23.]

4.5 Mallien linkittäminen ja yhdistäminen

Mallien vaatimien kirjoitettujen rivien määrä on yleensä moninkertainen verrattuna niiden kuvaamaan JSON-objektiin. Alla oleva malli kuvaa näennäisen yksinkertaista objektia:

```
"packet": {
  "payload": 211134,
  "encrypt": "sha256"
}
```

Objektia voisi kuvastaa seuraava malli:

```
  "description": "Packet for payload and encryption algorithm",
  "type": "object",
  "properties": {
    "description": "Payload as either string or integer",
    "payload": {
      "type": ["string", "integer"]
    },
    "encrypt": {
      "description": "Encryption algorithm used for payload",
       "type": "string",
       "pattern": "^[A-Za-z0-9]+",
       "minLength": 4
     }
   }
```

Kyseisen mallin vaatima rivimäärä on lähes nelinkertainen itse objektiin verrattuna. Lisäksi käytännön sovelluksissa esiintyy usein samankaltaista tietoa, joita voidaan kuvata samoilla malleilla.

4.5.1 *$id, $ref*

Avainsanoilla *$id* ja *$ref* voidaan malleissa viitata toisiin malleihin sekä tiedosto- että objektitasolla. *$id* toimii mallin yksilöllisenä tunnisteena, *$ref*-avainsanaa käytetään puolestaan malliin viittaamiseen. *$id*-avainsana ei ole pakollinen viittauksia tehtäessä, mutta sen käyttö malleissa on suositeltavaa. *$ref*:n arvo on **URI**.

Aiempi malli olisi selkeästi jaoteltavissa kahteen osamalliin:

```
1. {
2.   "payload": {
3.     "$id": "extended-json-conf-tool/payload.schema.json",
4.     "type": ["string", "integer"]
5.   }
6. }
```

ja

```
1. {
2.   "encrypt": {
3.     "$id": "extended-json-conf-tool/encrypt.schema.json",
4.     "description": "Encryption algorithm used for payload",
5.     "type": "string",
6.     "pattern": "^[A-Za-z0-9]+$"
7.   }
8. }
```

Näiden osamallien avulla *packet*-objekti voitaisiin mallintaa seuraavasti:

```
1. {
2.   "packet": {
3.     "properties": {
4.       "payload": {"$ref": "#/payload"},
5.       "encrypt": {"$ref": "#/encrypt"}
6.     }
7.   }
8. }
```

Viittaamisen myötä mallin ylläpito helpottuu: viittausten kohteita voidaan vaihtaa helposti sekä osamalleja *payload* ja *encrypt* voidaan käyttää myös osana muita malleja.

Aiemmassa esimerkissä *$ref*-avaimet saavat arvoksi merkkijonon, joka alkaa *#*-merkillä. Merkki kuvaa JSON-osoitinta ja sillä voidaan kertoa, mistä haluttua resurssia tulisi etsiä [24]. Asettamalla osoitin merkkijonon alkuun, kerrotaan, että viitattavaa mallia tulee etsiä *packet*-mallin sisältävän tiedoston juuresta. Mallit, joihin viitataan muissa malleissa, on tapana sijoittaa *definitions*-nimisen objektin sisään. Esimerkin mukainen tiedosto voisi kokonaisuudessaan näyttää tältä (*payload*- ja *encrypt*-mallit on selkeyden vuoksi typistetty):

```
1. {
2.   "definitions": {
3.     "payload": { ... },
4.     "encrypt": { ... }
5.   },
6.   "packet": {
7.     "properties": {
8.       "payload": {"$ref": "#/definitions/payload"},
9.       "encrypt": {"$ref": "#/definitions/encrypt"}
10.      }
11.    }
12.  }
```

Jos *payload*- ja *encrypt*-mallit haluttaisiin kirjoittaa toiseen, esimerkiksi *definitions.json*-nimiseen tiedostoon, voitaisiin niihin viitata antamalla viittauksen arvoksi tiedoston nimi sekä siirtämällä JSON-osoitin osoittamaan haluttuun resurssiin:

```
1. "payload": {"$ref": "definitions.json#/payload"},
2. "encrypt": {"$ref": "definitions.json#/encrypt"}
```

[14.] [25.]

4.5.2 anyOf, oneOf, allOf, not

Joskus voi olla tarpeen joko sallia useamman mallin mukaiset objektit tai käyttää useampaa mallia validointiin. Avainsanalla *anyOf* voidaan listata useita malleja, joista objektin täytyy täyttää yhden vaatimukset. Malli, joka sallii vähintään neljän merkin pituiset merkkijonot sekä kokonaisluvut, joiden arvo on vähintään 1000, voisi *anyOf*-avainsanaa käyttäen olla seuraavan kaltainen: [26.]

```
1. {
2.   "anyOf": [
3.     {
4.       "type": "string",
5.       "minLength": 4
6.     },
7.     {
8.       "type": "integer",
9.       "minimum": 1000
10.      }
11.   ]
12. }
```

Avainsanat *oneOf* sekä *allOf* toimivat syntaksin puolesta samoin kuin *anyOf*, mutta rajoittavat sallittujen objektien joukkoa: *oneOf*-avainsanaa käytettäessä objektin tulee olla validi ainoastaan yhtä listassa annettua mallia vasten ja *allOf*-avainsanaa käytettäessä mallin tulee olla validi jokaista listassa annettua mallia vasten. On huomattava, että *allOf*-avainsanalla on mahdollista luoda malleja, joita vasten mikään objekti ei voi olla validi. Alla oleva malli määrittää, että objektin täytyisi olla yhtä aikaa sekä merkkijono että kokonaisluku: [27.] [28.]

```
1. {
2.   "allOf": [
3.     {"type": "string"},
4.     {"type": "integer"}
5.   ]
6. }
```

not-avainsanalla voidaan julistaa malli, jota vasten objekti ei saa olla validi. Seuraava malli sallii kaikki sellaiset objektit, jotka eivät ole tyyppiä *integer* ja arvoltaan välillä 10-20: [29.]

```
1. {
2.   "not": [
3.     {
4.       "type": "integer",
5.       "minimum": 10, "maximum": 20
6.     }
7.   ]
8. }
```

5 Työkalun toteutuneet ominaisuudet

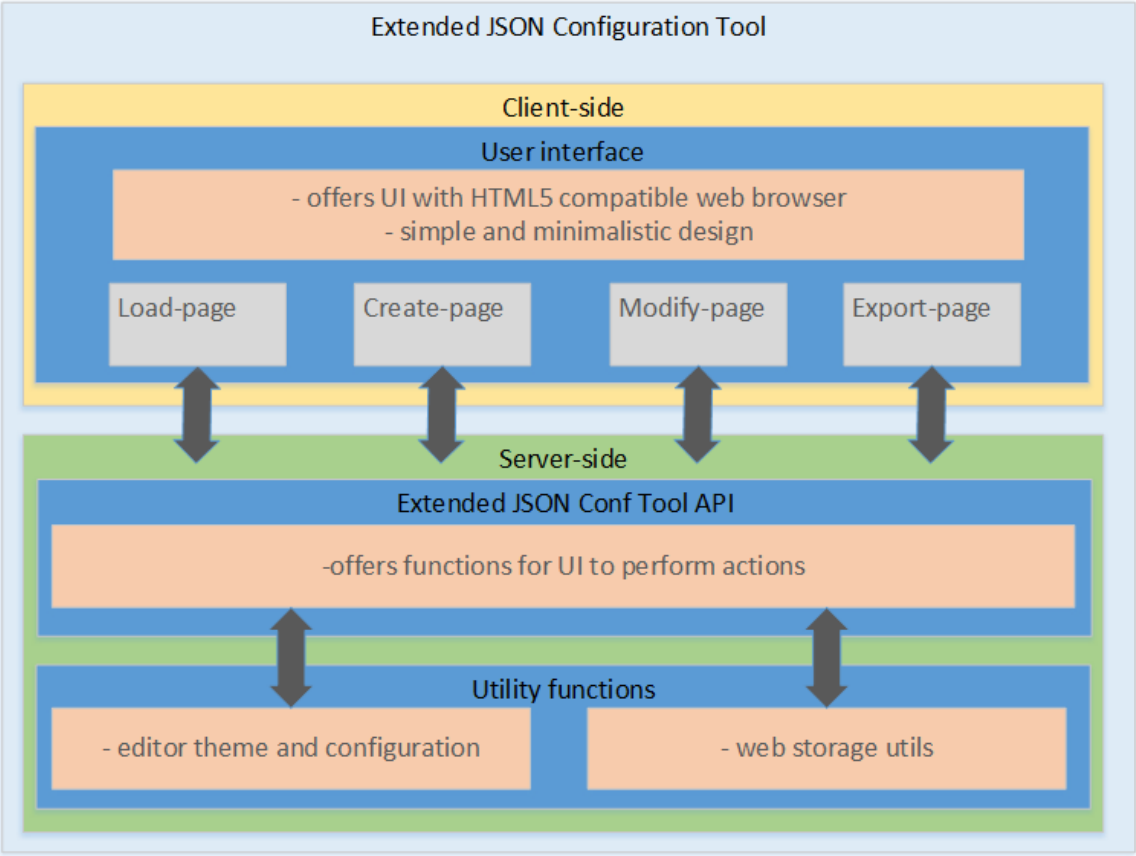
Työkaluun saatiin implementoitua kaikki toimeksiantajan vaatimusten mukaiset ominaisuudet. Ominaisuudet sekä niiden käyttö esitellään kohdassa 5.3 (Toiminnalliset ominaisuudet). Liiketoimintakerroksen arkkitehtuuri muuttui huomattavasti yksinkertaisemmaksi implementointivaiheessa sekä koodin sisältämien funktioiden että käyttöliittymän tarpeiden selkeytyessä. Käyttöliittymän eriyttäminen liiketoimintakerroksesta onnistui hyvin. Käyttöliittymän koodissa viitataan liiketoimintakerroksen logiikkaan ainoastaan ohjelmointirajapinnan kautta.

5.1 Toteutunut arkkitehtuuri sekä kansiorakenne

Liiketoimintakerroksen arkkitehtuuri koki implementointivaiheessa muutoksia: neljän erillisen komponentin sijasta liiketoimintakerroksen ohjelmointirajapinta sijoitettiin yhteen tiedostoon, yhden funktion alle. Tällaiseen toteutukseen päädyttiin, koska implementoinnin alkuvaiheessa kävi selväksi, että perustoiminnallisuuden saavuttamiseksi tarvittava sisäinen toiminnallisuus oli iteratiivisen toteutuksen sekä testauksen kannalta järkevin sijoittaa mahdollisimman tiiviisti. Liiketoimintakerroksen tarvitsemia apufunktioita eriytettiin omiin tiedostoihin. Arkkitehtuuri yksinkertaistui, kun liiketoimintakerroksen koodi sijoitettiin yhteen komponenttiin.

Projektin kansiorakenne muodostui seuraavasti:

app	Päätason kansio
|_ components	Vue.js-kirjastolla kirjoitetut yleiskomponentit
|_ js	Kolmannen osapuoltenKirjastojen lähdekoodit
|_ scripts	Työkalun liiketoimintakerroksen sekä käyttöliittymän koodi
|_ API	Työkalun ohjelmointirajapinnan koodi
|_ common	Yleisesti käytettävissä oleva koodi
|_ confs	Työkalun konfiguraatioon liittyvä koodi
|_ init	Alustukseen tarkoitettu koodi
|_ styles	CSS-tiedostot
|_ temp	Kehityksen tueksi tarkoitetuille tiedostoille
|_ json_files	Kehityksen aikana käytettävät JSON-tiedostot sekä mallit
index.html	Käyttöliittymän HTML-kuvaus



Kuva 4. Kuvaus toteutuneesta arkkitehtuurista

5.2 Käyttöliittymä sekä käytettävyys

Ohjelmointiosuuden alkuvaiheilla käytettävyyteen ei juuri panostettu. Tärkeintä oli saada tehtyä liiketoimintakerroksen koodissa erilaisia kokeiluja. Kun käyttöliittymää alettiin siirtämään suunnitelmista koodin puolella, huomattiin, että käyttöliittymän käyttö toi esiin sellaisia käyttötapauksia, joita ei suunnitelmista oltu osattu ottaa huomioon. Esimerkkinä mainittakoon mahdollisuus paljaan JSON-formaattisen tekstin näyttämiseen editorisivulla. Nämä huomiot toivat jonkin verran lisätöitä, mutta paransivat sekä liiketoimintakerroksen ohjelmointirajapintaa että itse käyttöliittymää.

Käyttöliittymän rakentaminen osoittautui verrattain työlääksi. *JSON Editor* –kirjaston tuottamien HTML-elementtien tyylittely mahdollisimman yksinkertaisiksi sekä tarkoituksenmukaisiksi vaati lopulta oman, *JSON Editor*:n teemaa kuvaavasta *AbstractTheme*-luokasta perivän luokan kirjoittamista. Tämä mahdollisti yksityiskohtaisemman CSS-attribuuttien käytön sekä kustomoidun toiminnallisuuden esimerkiksi virheilmoituksia näytettäessä.

Ohjelman käytettävyys parantui huomattavasti implementaatio-osuuden loppua kohden. Koska aiempaa kokemusta *Javascript+HTML+CSS* –yhdistelmän käytöstä ohjelmoinnissa ei juuri ollut, oli käyttöliittymän elementtejä paranneltava sitä mukaa kuin kokemusta kertyi lisää. *Vue.js*-kirjaston mukaan ottaminen osoittautui hyväksi päätökseksi, koska se mahdollisti tarvittavan ymmärryksen kerryttyä nopeat kokeilut sekä parantelut. Lisäksi sen avulla dynaamisten elementtien teko oli erittäin helppoa.

Alun perin haasteelliseksi ajateltu näytön koon mukaan skaalautuva käyttöliittymä osoittautui yllättävän helpoksi toteuttaa. Skaalautuvuutta testattiin ja parannettiin tekemällä käyttökokeiluja eri näyttökoilla 14 tuumasta 27 tuumaan asti.

Värimaailman suunnittelu aloitettiin samalla kun alettiin muovaamaan käyttöliittymäelementtejä tarkoituksenmukaisiksi. Värimaailma päätettiin toteuttaa toimeksiantajan verkkosivujen värimaailmaa mukaillen. Verkkosivujen väripaletti ei riittänyt kattamaan työkalun tarpeita, joten palettia laajennettiin käyttämällä apuna internetistä löytyviä väripalettigeneraattoreita. Värimaailmalla pyrittiin saavuttamaan tilanne, jossa käyttäjä voi heti nähdä minkä tyyppistä JSON-elementtiä hän on muokkaamassa.

5.3 Toiminnalliset ominaisuudet sekä käyttö

Seuraavissa kuvissa sekä niiden kuvauksissa taustoitetaan toiminnallisten ominaisuuksien implementointia käyttöliittymään sekä niiden käyttöä käyttöliittymältä. Pääpaino on konfiguraation muokkaukseen keskittyvällä sivulla. Esimerkkitiedostona ja -mallina käytetään *Simple Network Management Protocol* –standardin kuvitteellista, yhtä käyttäjää kuvaavaa tiedostoa sekä mallia. Mallilla on neljä avain-arvo-paria, joista kaksi on vaadittuja. Toiminnallisuuden käsittely pysyy selkeyden vuoksi yksinkertaisen JSON-mallin ympärillä. Listojen toimintaa kuvataan erillisen esimerkin avulla.

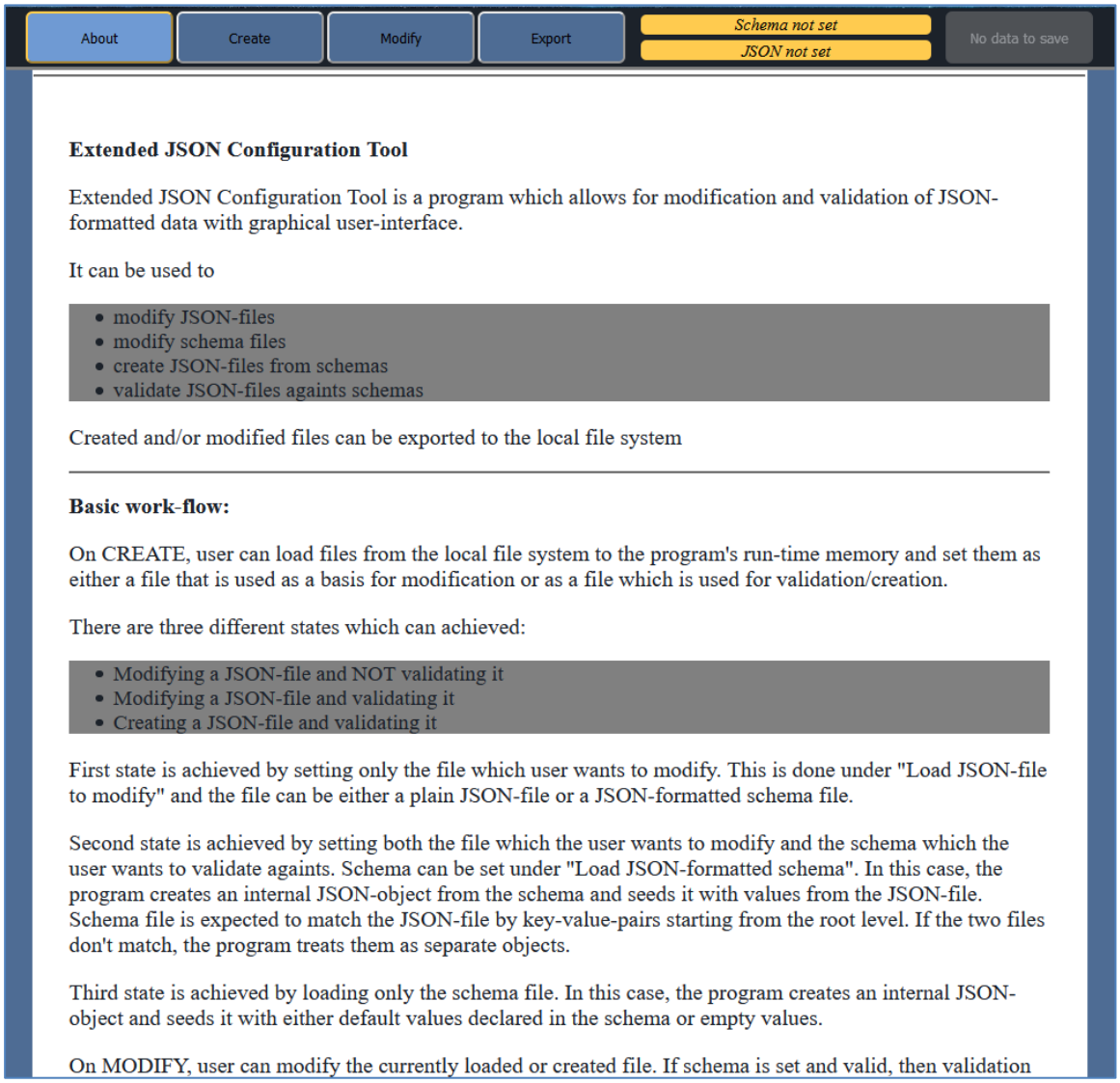
Esimerkkimallin mukainen JSON-objekti:

```
1. {
2.   "credentials": {
3.     "user_id": 1,
4.     "user_name": "myUserName",
5.     "auth_pass": "myAuthPass",
6.     "cryp_pass": "myCrypPass"
7.   }
8. }
```

Esimerkkimalli ei salli ylimääräisiä arvoja.

5.3.1 Aloitussivu sekä navigointipalkki

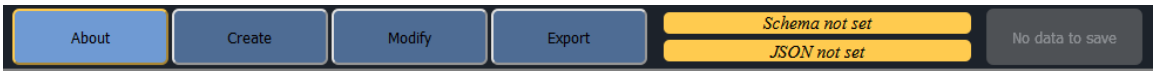
Työkalun aloitussivu on sijoitettu *About*-otsikon alle ja se sisältää lyhyen kuvauksen ohjelmasta sekä ohjeet eri käyttötapauksia varten. Aloitussivun kuvauksesta on nähtävissä myös ohjelman yleinen rakenne käyttöliittymän suhteen: Yläreunassa sijaitsevat navigointipainikkeet ohjelman eri osioihin, joista aktiivinen näytetään korostettuna. Navigointipainikkeiden oikealla puolella sijaitsevat alkutilanteessa keltaisella korostetut inforuudut sekä harmaalla korostettu tallennuspainike, joka aktivoituu, kun muokattavaan tiedostoon tehdään muutoksia.



Kuva 5. Aloitussivu

Navigointipalkki on staattinen ja pysyy paikoillaan, vaikka käytössä oleva sivu pitenee yli näytön korkeuden. Navigointipalkin *About*-, *Create*-, *Modify*- sekä *Export*-otsikoilla varustetut painikkeet vievät käyttäjän ohjelman eri toiminnallisuuksien pariin: *About*-painike vie aloitussivulle, *Create*-painikkeella pääsee sivulle, jolla voi ladata tiedostoja ohjelman muistiin, *Modify*-painikkeella päästään muokkaamaan konfiguraatiota ja *Export*-painikkeen takaa löytyvä toiminnallisuus mahdollistaa käsittelyn kohteena olevan konfiguraation tallentamisen käyttäjän tiedostojärjestelmään. Inforuudut ilmaisevat ohjelmaan tilan eivätkä ne ole klikattavissa. Ylempi palkki kertoo, onko ohjelmassa käytössä käyttäjän asettama JSON-malli ja alempi palkki kertoo, mikä on muokattavan JSON-datan status. Inforuudut vaihtavat väriä keltaisen ja vihreän välillä riippuen työkalun tilasta. Infopalkkien väristatuksia hyväksikäyttäen työkalun käyttäjän on

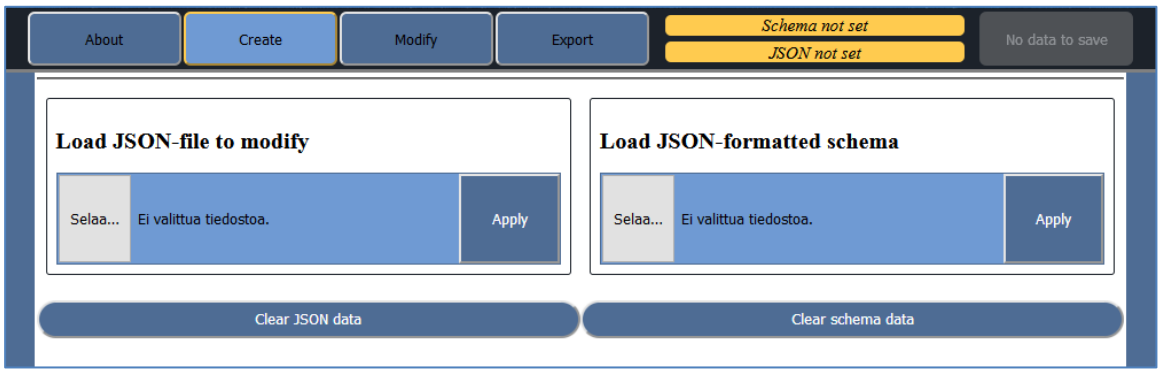
mahdollista päätellä sen nykyinen tila sekä se millaista käyttötapausta työkalu käsittelee. Väristatuksista kerrotaan syvällisemmin *Create*-painikkeen alta löytyvän toiminnallisuuden kuvauksessa. Navigointipalkin oikeaan laitaan on sijoitettu tallennuspainike, jolla saa tallennettua konfiguraatioon tehdyt muutokset verkkoselaimen väliaikaiseen muistiin. Painike aktivoituu, kun käyttäjä tekee konfiguraatioon muutoksia. Tallennuspainikkeen aktivoituessa sen taustaväri muuttuu keltaiseksi. Kun käyttäjä tallentaa muutokset painiketta klikkaamalla, taustaväri muuttuu tumman vihreäksi.



Kuva 6. Navigointipalkki

5.3.2 *Create*-sivu

Create-sivu mahdollistaa konfiguraatio- sekä mallitiedostojen tuonnin ohjelman muistiin käyttäjän tiedostojärjestelmästä sekä niiden poistamisen ohjelman muistista. JSON-muotoisten tiedostojen sekä mallien lataamiselle ja poistamiselle on toteutettu omat erilliset osiot.

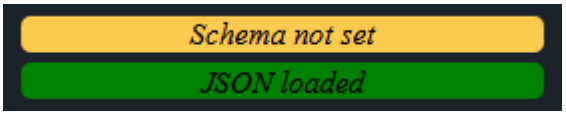


Kuva 7. *Create*-sivu sekä lataus- ja poisto-osiot

Ohjelman eri käyttötapaukset saadaan toteutettua varioimalla tiedostojen latausta näitä osioita hyväksikäyttämällä.

Tiedoston muokkaus ilman validointia

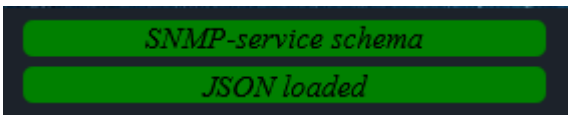
Jos käyttäjä haluaa vain muokata JSON-pohjaista tiedostoa ilman validointia, olkoon muokattava tiedosto JSON- tai malli-tiedosto, valitsee hän muokattavan tiedoston *Load JSON-file to modify* –otsikon alta löytyvän *Selaa*-painikkeen avulla sekä painaa *Apply*-painiketta. Tällöin alempi infopalkki muuttuu vihreäksi ja muuttaa tekstisisällön ilmaisemaan, että JSON-tiedosto on ladattu. Samalla tiedoston sisältämä JSON-objekti ilmestyy muokattavaksi *Modify*-välilehden alle.



Kuva 8. Infopalkin näkymä pelkän muokkauksen yhteydessä

Tiedoston muokkaus ja validointi

Jos käyttäjä haluaa muokkaamisen lisäksi varmentaa tiedoston, hän lataa ja hyväksyy muokattavan tiedoston lisäksi mallin, jota vasten varmennus tapahtuu. Mallin valitseminen ja hyväksyminen tapahtuu *Load JSON-formatted schema* –otsikon alta löytyvällä painikkeilla. Tällöin molemmat infopalkit ovat vihreitä ja lisäksi ylempi palkki sisältää mallin ylimmän tason *title*-avainsanan arvon, jos sellainen on annettu. Samalla *Modify*-sivu päivittyy sisältämään näkymän muokattavasta JSON-objektista täydennettynä mallin sisältämällä tiedolla.

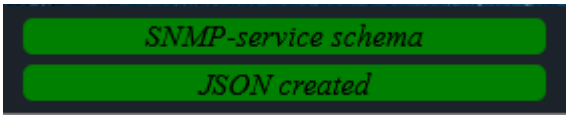


Kuva 9. Infopalkin näkymä muokkauksen ja validoinnin yhteydessä

Kun käyttäjä lataa ensin JSON-tiedoston ja sen jälkeen mallin, ohjelma luo sisäisesti ensin JSON-objektin mallin pohjalta, jonka jälkeen ohjelma alustaa JSON-objektin JSON-tiedoston arvoilla. Tämän vuoksi muokattavan tiedoston sekä mallin täytyy olla yhteensopivia. Tämä tarkoittaa sitä, että muokattavan tiedoston täytyy noudattaa mallin rakennetta sekä toisinpäin. Jos ohjelmaan ladataan objektia *Y* kuvaava tiedosto muokattavaksi ja malliksi valitaan objektin *Z* varmennukseen tarkoitettu malli, ohjelma käsittelee tiedostot erillisinä kokonaisuuksina ja näyttää *Modify*-sivulla sekä tiedoston että mallin datan erillisinä osioina.

Tiedoston luonti mallin pohjalta

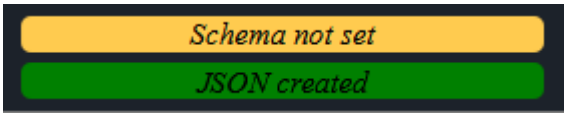
Jos käyttäjä haluaa luoda täysin uuden JSON-tiedoston mallin pohjalta, tulee hänen ladata ohjelmaan ainoastaan malli, jonka pohjalta tiedosto halutaan luoda. Tällöin molemmat infopalkit muuttuvat vihreäksi. Ylempi sisältää mahdollisen mallille annetun otsikon ja alempaan ilmestyy teksti kuvaamaan tilannetta, jossa JSON-objekti on luotu mallin pohjalta.



Kuva 10. Infopalkin näkymä luotaessa JSON-tiedosto mallin pohjalta

Kun käyttäjä lataa ainoastaan mallin, ohjelma luo JSON-objektin mallin pohjalta. Jos avaimen arvolle on mallissa annettu *default*-avainsanalla jokin arvo, käytetään tuota arvoa. Jos vakioarvoa ei ole annettu, käytetään merkkijonojen yhteydessä tyhjää merkkijonoa (”“) ja muiden arvojen kohdalla arvo jätetään täyttämättä.

Jos käyttäjä haluaa luoda JSON-tiedoston mallin pohjalta, mutta muokata sitä ilman varmennusta, tulee hänen ladata ensin ainoastaan malli, jonka jälkeen hänen tulee poistaa malli ohjelman muistista. Tämä onnistuu *Clear schema data*–painikkeella. Näin toimittaessa ohjelma luo ensin muokattavan JSON-objektin mallin pohjalta, mutta poistaa itse mallin käytöstä. Tällöin infopalkit kuvaavat tilannetta, jossa JSON-objekti on luotu, mutta mallia ei ole asetettu.



Kuva 11. Infopalkin näkymä poistettaessa malli JSON-objektin luonnin jälkeen

5.3.3 *Modify*-sivu

Modify-sivulla käyttäjä muokkaa asettamaansa tiedostoa. Sivun ylälaidassa on ilmaistu värikoodein minkälaista taustaväriä eri tyypit käyttävät: objektit kuvataan *Object*-otsikon ja listat *Array*-otsikon taustavärillä. Muut tyypit kuvataan *Value*-otsikon taustavärillä.

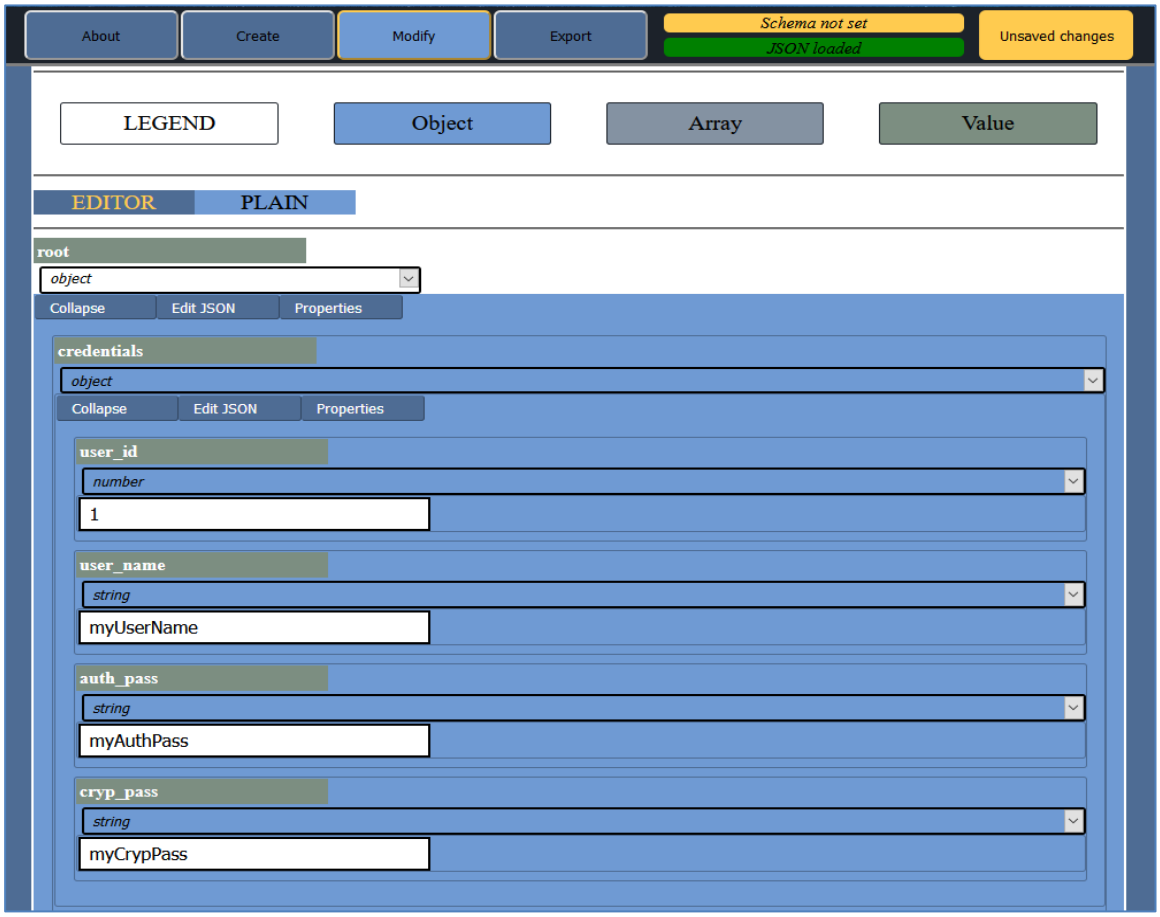
Kuva 12. Tyyppien värikooditus

Tyyppien värikooditusta seuraa valintapalkki, jolla käyttäjä voi valita haluaako hän muokata tiedostoa vai tarkastella tiedostoa tekstimuodossa. Aktiivinen näkymä on korostettu keltaisella tekstillä sekä tummemmalla taustavärillä.

Kuva 13. Näkymän valinta

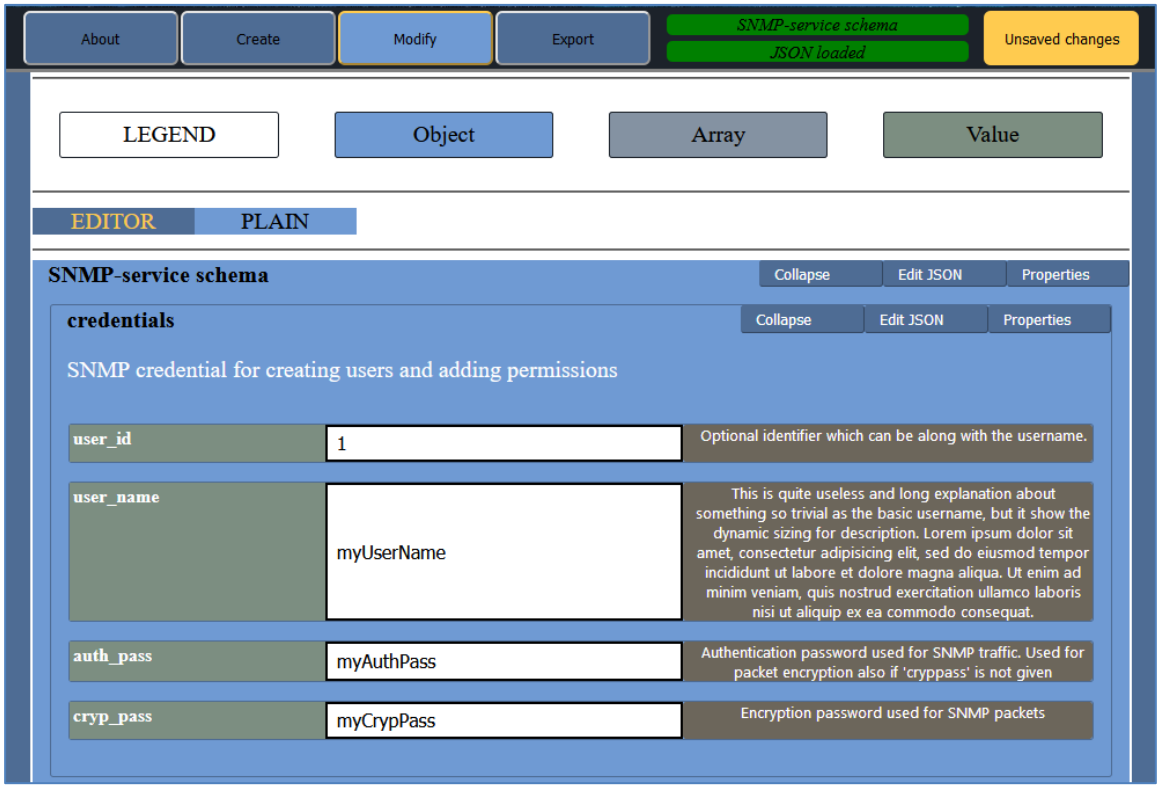
Valintapalkin ollessa *EDITOR*-tilassa, käyttäjä näkee editorinäkymän, jossa on mahdollista muokata tiedostoa. *PLAIN*-tilassa käyttäjä näkee tiedoston JSON-formaatissa merkkijonona. *PLAIN*-tilassa ei ole mahdollista muokata tiedostoa.

Ladattaessa pelkkä JSON-tiedosto ei editorinäkymässä näytetä malleihin liittyviä ominaisuuksia. Käyttäjä voi muokata tiedostoa haluamallaan tavalla.

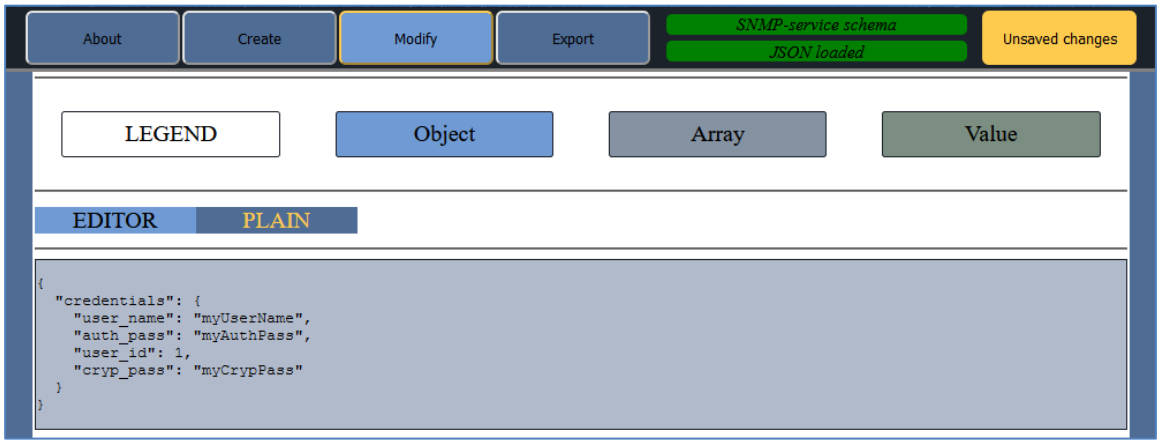


Kuva 14. Editorinäkymä ilman mallia

Jos käyttäjä lataa mallin, editorinäkymässä näytetään mallin tarjoamaa tietoa, kuten avain-arvo-parien mahdolliset kuvaukset ja otsikot.

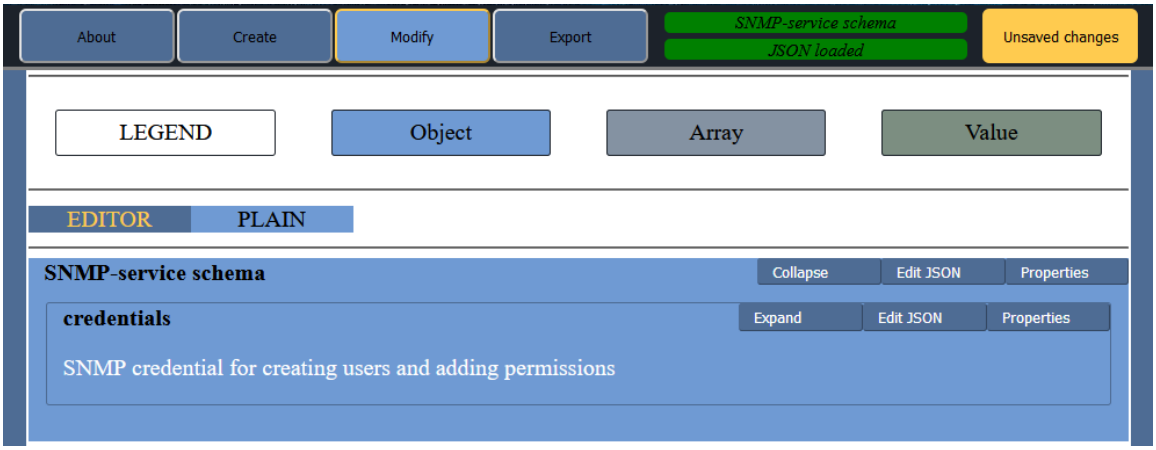


Kuva 15. Editorinäkymä, kun malli on asetettu



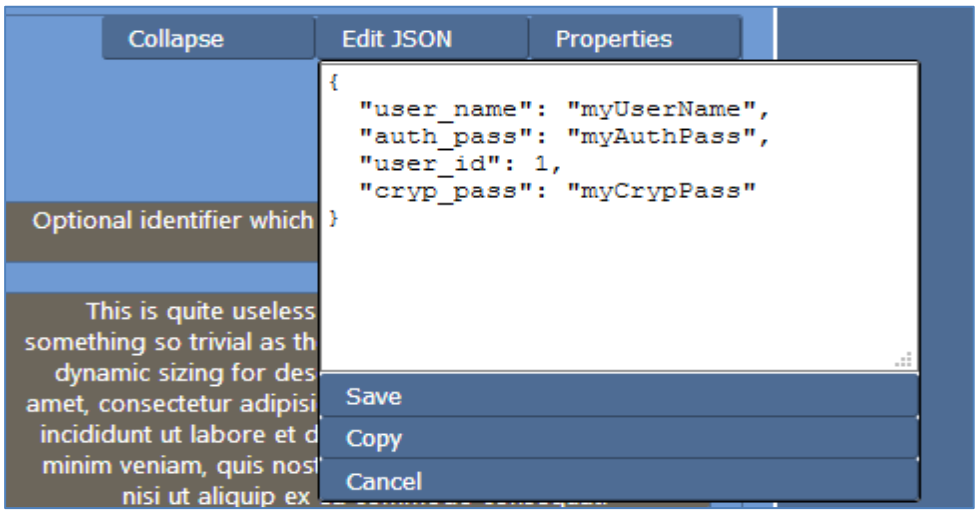
Kuva 16. JSON-tiedoston tarkastelunäkymä *PLAIN*-tilassa

Editorinäkymässä näkyvät *Collapse*, *Edit JSON* ja *Properties* –painikkeet ovat objektikohtaisia. *Collapse*-painikkeella käyttäjä voi piilottaa haluamansa objektin näkymän. Tällä tavoin suurien tiedostojen näkymää on helpompi hallita. Näkymän ollessa piilotettuna painikkeen otsikko on *Expand*.



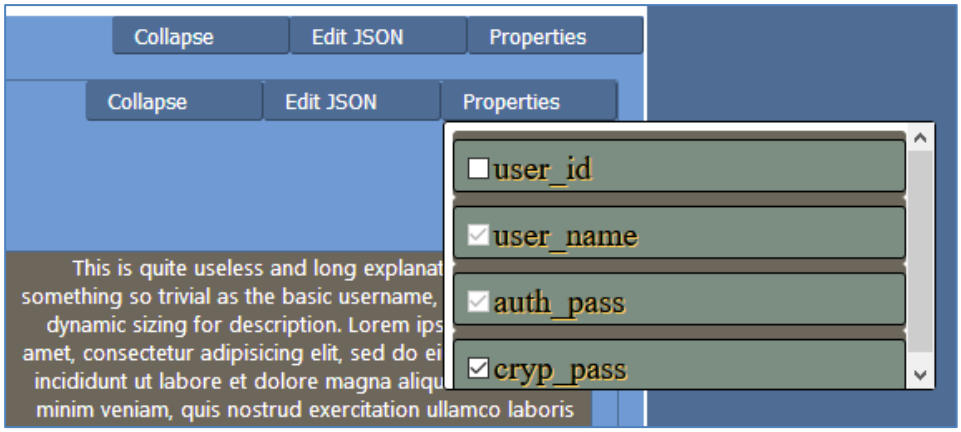
Kuva 17. *credentials*-objektin näkymä piilotettuna

Edit JSON–painikkeella voidaan editoida haluttua objektia suoraan tekstiformaatissa. Painike aukaisee pienen tekstieditorin, jossa objektin arvoja voi muokata. Muutokset on mahdollista tallentaa tai hylätä ja teksti on mahdollista kopioida käyttöjärjestelmän leikepöydälle. On huomattava, että mikäli malli on käytössä, ei *required*-avainsanalla julistettuja avain-arvo-pareja voi poistaa ja jos malli ei salli ylimääräisiä arvoja asettamalla avainsana *additionalProperties* arvoon epätosi (*false*), ei ylimääräisten avainten luonti vaikuta muokattavaan tiedostoon millään lailla.

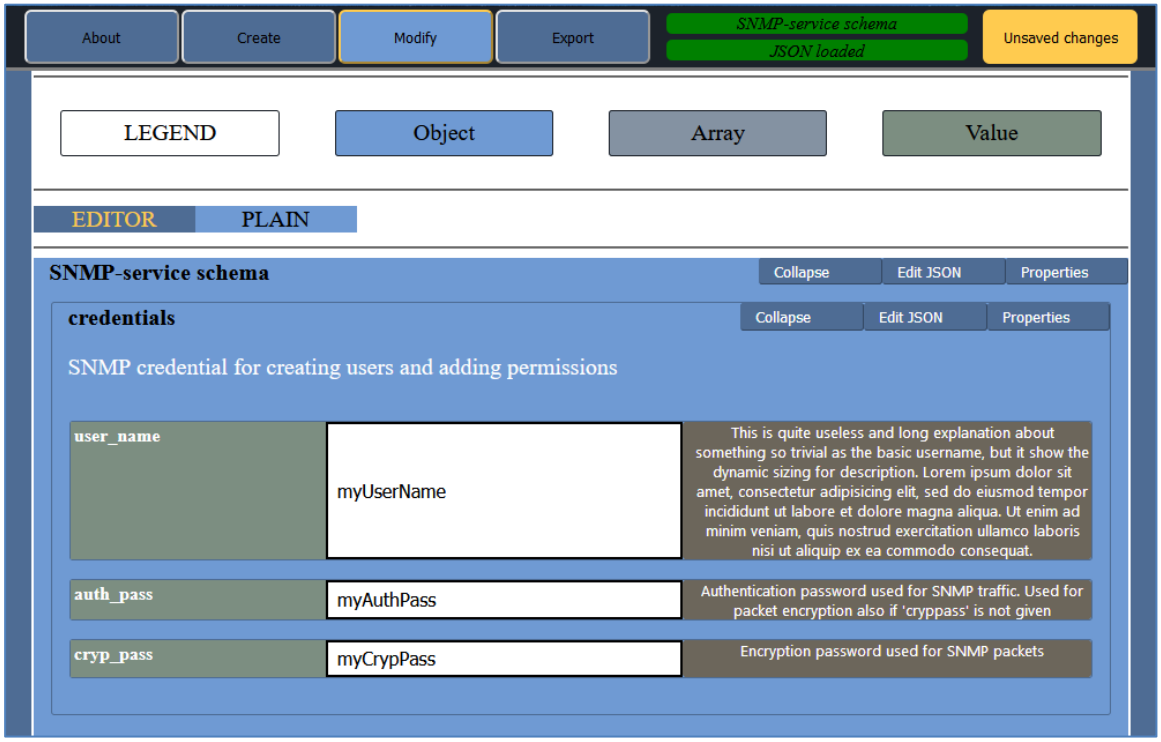


Kuva 18. *credentials*-objektin tekstimuokkausnäkymä

Properties-painikkeella voidaan vaikuttaa siihen, mitä ominaisuuksia muokattavassa tiedostossa on käytössä. Esimerkkimallissa on kaksi vaadittua ominaisuutta sekä kaksi haluttaessa käytettävää. Painike aukaisee näkymän, jossa näkyvät objektin kaikki ominaisuudet. Pakollisten ominaisuuksien valintalaatikot näkyvät hieman läpinäkyvinä eikä niitä voi ottaa pois päältä. Muut ominaisuudet voidaan asettaa päälle tai ottaa pois päältä.



Kuva 19. *credentials*-objektin ominaisuudet *user_id* poistettuna



Kuva 20. Editorinäkymä ilman *user_id*-ominaisuutta

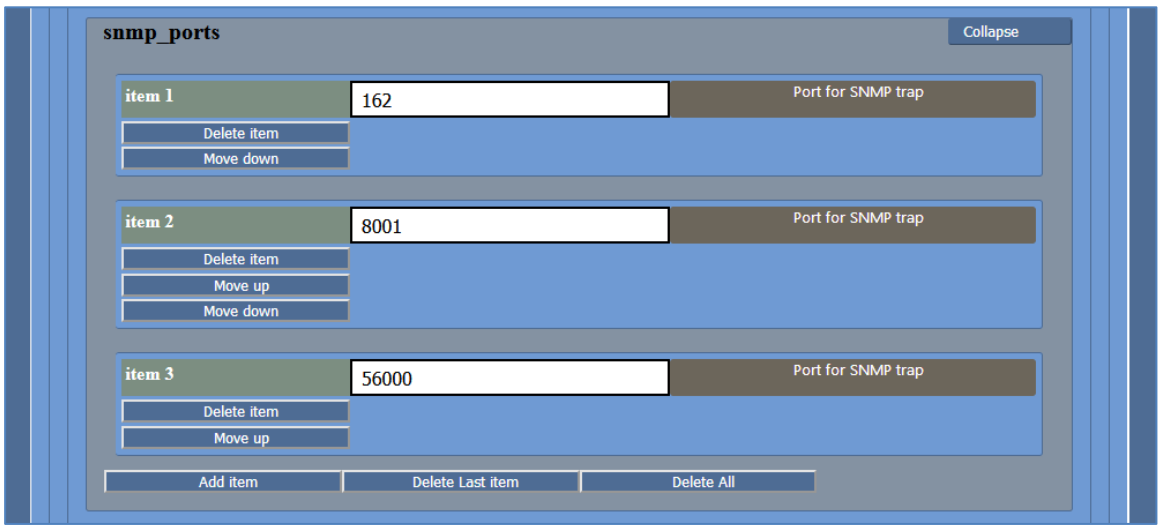
Objektien ja listojen sisältämät paljaat arvot esitetään kolmen elementin kokonaisuuksina. Ensimmäinen elementti sisältää ominaisuuden nimen, toinen sisältää ominaisuuden arvon ja kolmas sisältää ominaisuuden kuvauksen tai virheilmoituksen virheellisestä arvosta.

Kuva 21. Paljaiden arvojen näkymä

Ylläolevassa kuvassa *used_id* on saanut arvon, joka on mallin mukaisesti liian pieni. Arvon kuvauksen tilalla näytetään virheilmoitus. *user_name*, *auth_pass* sekä *cryp_pass* –ominaisuudet käyttävät *pattern*-avainsanalla julistettua säännöllistä lauseketta (**regex**) rakenteen varmentamiseen. Kuvan esimerkissä sekä *user_name* että *auth_pass* saavat virheelliset arvot.

Listat

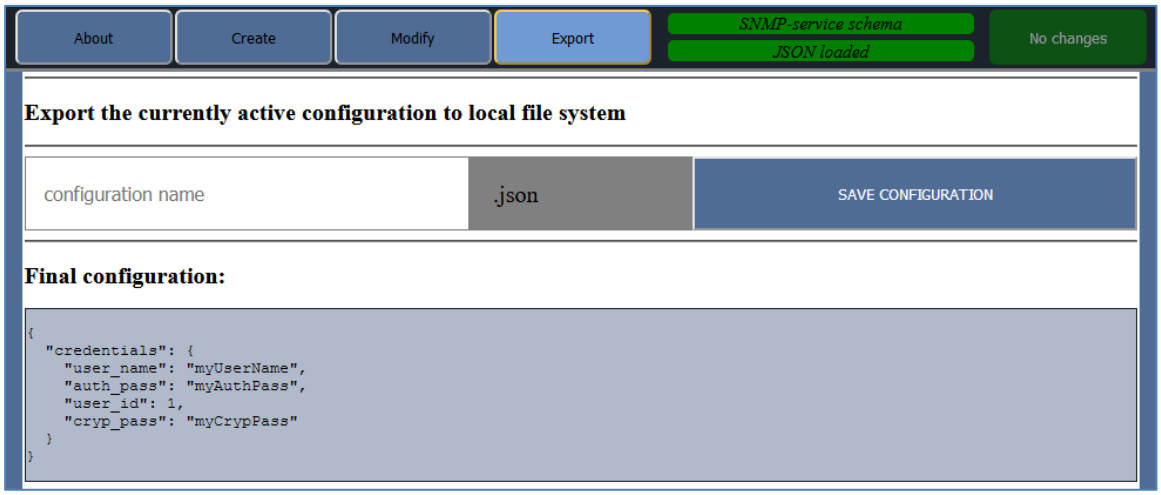
Editorinäkymässä listoihin voidaan lisätä elementtejä, poistaa niitä sekä vaihtaa elementtien järjestystä. Yksittäistä elementtiä käsitellään tyypin mukaisesti. Jos lista kuvaa monikkoa, ovat elementit monikon mukaisessa järjestyksessä.



Kuva 22: Esimerkki listojen elementeistä

5.3.4 *Export*-sivu

Kun käyttäjä on muokannut tiedostosta halutun kaltaisen, pystyy hän siirtämään tiedostojärjestelmään käyttäen *Export*-sivun tarjoamaa tallennusmahdollisuutta. Sivulla on mahdollista antaa tiedostonimi, joka saa automaattisesti *.json*-päätteen, sekä tallentaa tiedosto. Lisäksi sivulla näytetään käyttäjälle lopullinen tiedoston sisältö samaan tapaan kuin *Modify*-sivulla *PLAIN*-tilassa.



Kuva 23: *Export*-sivun näkymä

Konfiguraation nimi on pakollista antaa, muussa tapauksessa työkalu ilmoittaa puuttuvasta nimestä eikä mahdollista tallentamista.

Tallennettaessa työkalu, verkkoselaimen oletusasetuksista riippuen, joko tallentaa tiedoston selaimen asetusten mukaiseen kansioon tai kysyy käyttäjältä, haluaako hän tallentaa tiedoston vai avata sen jollain ohjelmalla.

5.4 Jatkokehitys

Työkalun kehityksen edetessä ilmeni sekä toiminnallisia että visuaalisia parannuksia, jotka siihen tulisi jatkokehityksessä implementoida:

- Työkalun käytettävyyden helpottamiseksi täytyy käyttöliittymän rakennetta selkeyttää eri käyttötapauksiin pääsemisen osalta. Tiedostojen lataamiseen ja lataamatta jättämiseen perustuva käyttötapauksen saavuttaminen ei palvele käyttäjiä niin hyvin kuin esimerkiksi dialogiin perustuva tapa, jossa käyttäjälle annetaan ohjatusti selkeät raamit, joiden puitteissa toimia.

- Käyttöliittymän näkymien sekä ohjelman toiminnan muokkaamisen tulisi olla mahdollista erillisen asetusvalikon kautta.

- Käyttäjän tulisi pystyä selaamaan JSON-tiedoston rakennetta erillisen puunäkymän kautta. Tämä helpottaisi varsinkin isojen tiedostojen muokkausta.

- Käyttäjällä tulisi pystyä paikalliseen tiedostojärjestelmään tallentamisen lisäksi lähettämään konfiguraatiotiedosto suoraan sitä käyttävälle laitteelle esimerkiksi *SSH*-yhteyttä hyväksikäyttäen. Käyttäjän tulisi pystyä konfiguroimaan tiedostoa käyttävän laitteen saavutettavuusparametrin, kuten IP-osoite sekä salasana.

- Käyttäjän tulee pystyä määrittelemään se kansio, jota käytetään juurikansiona linkityksiä tehtäessä.

Ensimmäisenä jatkokehityskohteena on ajateltu olevan mahdollisuus lähettää konfiguraatiotiedostoja työkalusta suoraan laitteelle. Tämä helpottaisi toimeksiantajan tuotteiden parissa työskentelevien ohjelmistokehittäjien työtä ja takaisi työkalulle käyttötunteja ja siten jatkuvaa testausta sekä parannusehdotuksia.

6 Pohdinta

JSON-formaattia olevien tiedostojen varmennus *JSON Scheman* mukaisia malleja vastaan on selkeä tapa rajata JSON-objektien ominaisuuksia. Opinnäytetyön myötä kävi selväksi, että myös JSON-tiedostojen generointi mallien pohjalta on oiva tapa helpottaa konfiguraatioita tekevien ja tarvitsevien kehittäjien työtä. Mallit itsessään voivat olla pitkiä sekä monimutkaisia, paikoitellen jopa sekavia, ja niiden tulkinta voi vaatia *JSON Scheman* perusteellista ymmärtämistä. Toteutuneen työkalun avulla konfigurointiin käytettävän JSON-tiedoston voi luoda ja varmentaa mallien pohjalta ymmärtämättä itse malleista mitään.

JSON Schema–standardin tutkiminen tuotti hyvän perusosaamisen, jonka pohjalta laadittuja esimerkkejä voidaan käyttää perehdytykseen. Standardi on elävä ja sen sekä työkalussa käytettävän *JSON Editor*–kirjaston kehittymistä tulee seurata jatkokehityksen ohella.

Työkalusta muodostui implementaation loppua kohden käyttökelpoinen ohjelma. Selkeitä parannuskohteita on kuitenkin useita.

Lähtötilanteessa vajavaisen web-teknologia-osaamisen myötä kehitys oli paikoitellen hidasta sekä monia osia koodista piti kirjoittaa useaan kertaan. Lopputuloksena on kuitenkin kohtuu selkeä ja kehitettävä koodipohja, johon on jo olemassa jatkokehitysideoita. Jatkokehityksen kannalta on tärkeää saada työkalu yleiseen käyttöön sekä kerätä käyttäjäpalautetta.

LÄHTEET

[1] Bittium TACWIN
. Available at: https://www.bittium.com/tactical-communications/tactical-ip-networks. Accessed 23.3., 2019.

[2] Droettboom M. JSON Schema -implementations
. Available at: https://json-schema.org/implementations.html. Accessed 25.3., 2019.

[3] Vilmusenaho S. AaltoWiki. 2011; Available at: https://wiki.aalto.fi/pages/viewpage.action?pageId=56199997. Accessed 25.3., 2019.

[4] Dorn J, et. al. JSON Editor - Github
. Available at: https://github.com/json-editor/json-editor. Accessed 30.3., 2019.

[5] jQuery Foundation. jQuery Javascript-kirjasto
. Available at: https://jquery.com/. Accessed 30.3., 2019.

[6] You E. Vue.js Javascript-kirjasto
. Available at: https://vuejs.org/. Accessed 30.3., 2019.

[7] Droettboom M. JSON Schema - object
. Available at: https://json-schema.org/understanding-json-schema/reference/object.html#object. Accessed 3.3., 2019.

[8] json.org. Introducing JSON
. Available at: https://www.json.org/. Accessed 23.3., 2019.

[9] json-schema.org. JSON Schema
. Available at: https://json-schema.org/. Accessed 10.3., 2019.

[10] Andrews H, Wright A. JSON Schema: A Media Type for Describing JSON Documents. Available at: https://tools.ietf.org/id/draft-handrews-json-schema-00.html. Accessed 23.3., 2019.

[11] IETF Tools json-schema-draft-01
. 2018; Available at: https://tools.ietf.org/html/draft-handrews-json-schema-01. Accessed 24.2., 2019.

[12] Wright A, Andrews H, Luff G. JSON Scema validation draft
. Available at: https://json-schema.org/latest/json-schema-validation.html. Accessed 10.3., 2019.

[13] Droettboom M. JSON Schema - $schema
. Available at: https://json-schema.org/understanding-json-schema/reference/schema.html#schema. Accessed 17.3., 2019.

[14] Droettboom M. JSON Schema - $id
. Available at: https://json-schema.org/understanding-json-schema/structuring.html#id. Accessed 18.3., 2019.

[15] Droettboom M. JSON Schema - annotations and comment
. Available at: https://json-schema.org/understanding-json-schema/reference/generic.html#annotations. Accessed 18.3., 2019.

[16] Droettboom M. JSON Schema - type
. 2019; Available at: https://json-schema.org/understanding-json-schema/basics.html#the-type-keyword. Accessed 10.3., 2019.

[17] Droettboom M. JSON Schema - enum
. Available at: https://json-schema.org/understanding-json-schema/reference/generic.html#enumerated-values. Accessed 18.3., 2019.

[18] Lahtonen T. Tuple
. Available at: http://appro.mit.jyu.fi/tiea2080/luennot/python/#TOC14. Accessed 16.3., 2019.

[19] Droettboom M. JSON Schema - array
. Available at: https://json-schema.org/understanding-json-schema/reference/array.html#array. Accessed 3.3., 2019.

[20] Droettboom M. JSON Schema - string
. Available at: https://json-schema.org/understanding-json-schema/reference/string.html#string.

[21] Droettboom M. JSON Schema - numeric types
. Available at: https://json-schema.org/understanding-json-schema/reference/numeric.html#numeric-types. Accessed 13.3., 2019.

[22] Droettboom M. JSON Schema - boolean
. Available at: https://json-schema.org/understanding-json-schema/reference/boolean.html#boolean. Accessed 13.3., 2019.

[23] Droettboom M. JSON Schema - null
. Available at: https://json-schema.org/understanding-json-schema/reference/null.html#null. Accessed 13.3., 2019.

[24] Bryan P, Zyp K, Nottingham M. JSON pointer
. 2013; Available at: https://tools.ietf.org/html/rfc6901. Accessed 23.3., 2019.

[25] Droettboom M. JSON Schema - using $id with $ref
. Available at: https://json-schema.org/understanding-json-schema/structuring.html?highlight=definitions#using-id-with-ref. Accessed 6.4., 2019.

[26] Droettboom M. JSON Schema - keyword anyOf
. Available at: https://json-schema.org/understanding-json-schema/reference/combining.html#anyof. Accessed 6.4., 2019.

[27] Droettboom M. JSON Schema - keyword allOf
. Available at: https://json-schema.org/understanding-json-schema/reference/combining.html#allof. Accessed 6.4., 2019.

[28] Droettboom M. JSON Schema - keyword oneOf
. Available at: https://json-schema.org/understanding-json-schema/reference/combining.html#oneof. Accessed 6.4., 2019.

[29]Droettboom M. JSON Schema - keyword not
. Available at: https://json-schema.org/understanding-json-schema/reference/combining.html#not. Accessed 6.4., 2019.