Matias Mäkelä

Koodigeneroinnin hyödyntäminen suunnittelujärjestelmäkirjaston kehitystyössä ja ylläpidossa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

2.3.2023

Tiivistelmä

Tekijä:	Matias Mäkelä
Otsikko:	Koodigeneroinnin hyödyntäminen suunnittelujärjestelmäkirjaston kehitystyössä ja ylläpidossa
Sivumäärä:	40 sivua
Aika:	2.3.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander Projektimanageri Samuli Mäkinen

---

Edistysaskeleet tekoälyn saralla ovat mahdollistaneet vanhojen työskentelytapojen tehostamisen. Koodigenerointi, eli ohjelmakoodin luominen ohjelmallisesti, mahdollistaa nopeamman koodin tuottamisen perinteiseen manuaaliseen koodaukseen verrattuna. Tässä opinnäytetyössä tutkittiin Animan tarjoamaa koodigenerointia ja sen hyötyä suunnittelujärjestelmän kehitystyössä sekä ylläpidossa. Opinnäytetyössä otettiin käyttöön minimalistinen Figmassa suunniteltu suunnittelujärjestelmä, jonka pohjalta toteutettiin suunnittelujärjestelmäkirjasto. Toteutus tehtiin manuaalisesti koodaamalla ja koodigeneroimalla, jotta näiden kahden tapaa pystyttiin lopuksi vertailemaan.

Vertailun tuloksena päädyttiin johtopäätelmiin, että Animalla koodigenerointi vaatii vaivannäköä suunnittelijalta sekä kehittäjältä. Jotta generoitu koodi olisi laadukasta, täytyy komponentit Figmassa määritellä tietyllä tavalla. Kun komponentit ovat oikein määriteltyjä, Anima generoi uskottavan näköisiä React-komponentteja, mutta niistä puuttuvat kaikki toiminnallisuudet, jotka kehittäjän täytyy manuaalisesti lisätä. Tällöin myös koodigeneroinnin tuoma hyöty kirjaston ylläpidossä liittyisi pelkästään ulkonäköasioihin. Näistä syistä ennen Animan koodigeneroinnin käyttöönottoa on syytä projektikohtaisesti arvioida käyttöönotosta johtuvat hyödyt sekä haitat.

Avainsanat: Koodigenerointi, Suunnittelujärjestelmä, Figma, React, Anima

Abstract

Author:	Matias Mäkelä
Title:	Utilization of Code Generation in Development and Maintenance of Design System Library
Number of Pages:	40 pages
Date:	2 March 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer Samuli Mäkinen, Project Manager

---

Advances in the field of artificial intelligence have made it possible to make old ways of working more efficient. Code generation, i.e., creating program code programmatically, enables faster code production in comparison to traditional manual coding. The study investigated the code generation offered by Anima and its usefulness in the development and maintenance of a design system. A minimalistic design system designed in Figma was taken into use, on the basis of which a design system library was implemented. The implementation was done by manual coding and code generation, so that the two methods could then be compared.

As a result of the comparison, it was concluded that code generation with Anima requires effort from the designer and the developer. In order for the generated code to be of high quality, the components in Figma must be defined in a certain way. When the components are properly defined, Anima generates React components that look believable, but they lack all the functionality that the developer then has to add manually. In that case, the benefit brought by code generation in library maintenance would be related only to the look of the said components. For these reasons, before taking Anima's code generation into use, it is necessary to evaluate its advantages and disadvantages on a project-by-project basis.

Keywords: Code generation, Design system, Figma, React, Anima

Sisällys

Lyhenteet

CSS: *Cascading Style Sheets*. Tekniikka, jolla määritellään ulkoasu HTML:llä tehdylle verkkosivulle.

HTML: *HyperText Markup Language*. Kuvauskieli, jolla rakennetaan verkkosivuja.

Sass: *Syntactically Awesome Stylesheet*. Esiprosessori komentokieli, joka tulkitaan tai käännetään CSS:äksi.

DOM: *Document Object Model*. Tapa kuvata rakenteisen dokumentin, kuten HTML:n rakenne puuna.

UI: *User interface*. Suomeksi käyttöliittymä, joka on laitteen, ohjelmiston tai minkä tahansa tuotteen osa, jonka kautta käyttäjä on vuorovaikutuksessa tuotteen kanssa.

1 Johdanto

Koodi ja erilaiset sovellukset ovat yhä suuremmassa osassa arkipäiviämme. John Backuksen vuonna 1954 luoma Fortran oli ensimmäinen korkean tason ohjelmointikieli (Bellis 2020). Siitä lähtien erilaisten ohjelmointikielten ja kirjastojen määrä on kasvanut räjähdysmäisesti. Kirjastot toimivat yli 21 miljoonan ohjelmistokehittäjän työkaluina vuonna 2016 (Evans Data Corporation 2017). Työ on kuitenkin vaativaa ja paikoitellen hidasta ja toisteista, mikä on johtanut automaattisen koodigeneroinnin mahdollisuuksien tutkimiseen, joka on ohjelmakoodin luomista ohjelmallisesti. Termillä koodigenerointi on myös muita merkityksiä, mutta tässä työssä sillä viitataan automaattiseen ohjelmointiin, eli lähdekoodin generointiin.

Tässä työssä tutkitaan koodigeneroinnin mahdollisuuksia selvitystyön muodossa. Suunnittelujärjestelmän (engl. Design System) ja sitä vastaavan suunnittelujärjestelmäkirjaston koodin ylläpito on työlästä. Suunnittelujärjestelmä sisältää valmiiksi määritellyt standardit, periaatteet sekä komponentit, jotka ovat valmiina käytettäviksi. Suunnittelujärjestelmää käyttäen käyttöliittymien toteutus- sekä suunnittelutyö on tehokkaampaa. (Ojala 2018.) Suunnittelujärjestelmäkirjaston koodin työläästä ylläpidosta aiheutuvaa työkuormaa voitaisiin vähentää koodigeneroinnin avulla, mikä toimi motivaattorina aiheen tutkimiseen. Työn tavoitteena oli tutkia, kuinka Figma-suunnittelutyökalulla suunnitellusta suunnittelujärjestelmästä saisi generoitua koodia käytettäväksi ohjelmistoprojekteissa ja onko generoitu koodi soveltuvaa käytettäväksi sellaisenaan.

Toisessa luvussa käsitellään koodigeneroinnin teoriaa ja tutustutaan erilaisiin työkaluihin. Kolmannessa luvussa tutustutaan tarkemmin käytettäviin menetelmiin. Neljännessä luvussa on esitelty itse toteutettu koodigeneroinnin tutkiminen, jonka jälkeen viidennessä luvussa esitellään tulokset. Lopuksi pohditaan koodigeneroinnin mahdollisuuksia ja tulevaisuudennäkymiä.

2 Koodigenerointi

2.1 Koodigeneroinnin historia

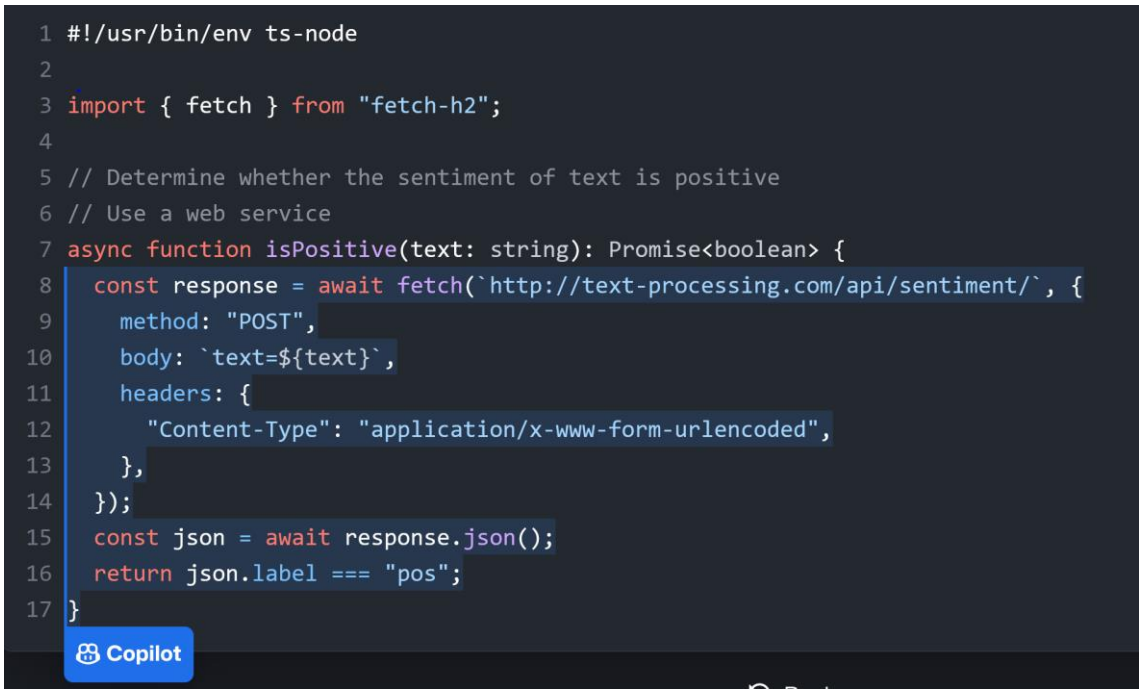
Automaattinen ohjelmointi eli koodigenerointi on syntynyt halusta uudelleen käyttää koodia, ja saada tietokone tekemään ihmisen työ (Hui ja Chun 2004). Automaattisen ohjelmoinnin käsite, kuten myös ohjelmointi on muuttunut paljon vuosien saatossa. Sillon kun käsitettä alettiin käyttää, sillä viitattiin kaikkiin niihin tapoihin, joilla pystyttiin tehostamaan koodin kirjoittajan tuottavuutta. Tietotekniikan alkuaikoina käsitteellä tarkoitettiin minkä vain järjestelmän mahdollisuutta automaattisesti luoda konekieltä. (Järvinen 2021, O'Neill ja Spector 2020, Barr ja Feigenbaum 1982.) Nykypäivän ohjelmointia voidaankin verrata siihen, minkälaista automaattinen ohjelmointi ennen oli (Hui ja Chun 2004). Yksi nykypäivän automaattisen ohjelmoinnin määritelmä tulee O'Neilliltä ja Spectorilta (2020), joiden mukaan se on tietokoneohjelman automaattinen luonti korkean tason tiedonsyötön perusteella. (O'Neill ja Spector 2020.)

Mildred Koss oli ohjelmoimassa ensimmäistä yhdysvaltalaista kaupallisesti valmistettua tietokonetta. Hänen mukaansa konekoodin kirjoittaminen sisälsi useita ikäviä vaiheita – prosessin jakamista erillisiin käskyihin, tiettyjen muistipaikkojen osoittamista komennoille ja I/O-puskurien hallintaa. Näiden matemaattisten rutiinien, kirjastojen ja ohjelmien lajittelun toteuttamisen jälkeen tehtävänä oli tarkastella laajempaa ohjelmointiprosessia. Täytyi ymmärtää, kuinka käyttää testattua koodia uudelleen ja saada koneen apua ohjelmoinnissa. Ohjelmoinnin yhteydessä tutkittiin prosessia ja yritettiin miettiä tapoja abstraktoida nämä vaiheet, jotta ne saataisiin sisällytettyä korkeamman tason kieleen, Koss kertoo. Tämä johti tulkkien, kokoajien, kääntäjien ja generaattoreiden kehittämiseen, jotka on suunniteltu toimimaan tai tuottamaan muita ohjelmia, eli toisin sanoen automaattiseen ohjelmointiin. (Hui ja Chun 2004.)

2.2 Koodigeneroinnin menetelmiä

Koodigenerointiin on useita eri menetelmiä. Yksi niistä on kääntäjä, joka generoi ohjelmoijan lähdekoodista konekoodia. Konekoodi on binääristen bittien eli ykkösten ja nollien sarja, joita tietokone tulkitsee operandeiksi ja komennoiksi (Britannica 2022). Myös koneoppimisen saralla on tutkittu erilaisia vaihtoehtoja tukemaan ohjelmoijan työtä ja generoimaan koodia. Yksi niistä on ongelmakeskeinen oppiminen (engl. Case-Based Reasoning). Ongelmakeskeistä oppimista voidaan käyttää ylläpitämään koodirutiinien kirjastoa, joka sisältää sellaisia koodirutiineja, jotka parhaiten vastaavat käyttäjän eli ohjelmoijan tarpeita. Tekoäly voi myös tarjota niitä ohjelmoijalle. Vaihtoehtoisesti geneettisen ja evoluutio-ohjelmoinnin avulla koodia voidaan mutatoida ja testata parannuksia varten. Jos koodia parannetaan, se voi toimia perustana seuraavan sukupolven koodia varten. Geneettinen ohjelmointi on biologisen evoluution kaltainen evoluutioalgoritmiin perustuva menetelmä, jolla käyttäjän määrittelemän tehtävän suorittamiseen löydetään tietokoneohjelmia. Evoluutioalgoritmi on tietokoneohjelman luomistapa, jossa hyödynnetään evoluutioteorian mukaisesti. Luonnonvalinta valitsee sopivimmat mutaatiot eli perimän muutokset jatkoon. (Danilchenko et. al 2012.)

Yksi esimerkki tekoälyllisestä koodigeneroinnista on Github Copilot, joka voidaan asentaa lisäosana esimerkiksi VSCodeen, joka on koodieditori, jolla ohjelmoija voi kirjoittaa koodia. Github Copilot tarjoaa koodiehdotuksia koneoppimismallista, jonka OpenAI on rakentanut miljardeista avoimen lähdekoodin koodiriveistä. Github Copilot toimii ikään kuin ohjelmoijan parina, joka tarjoaa koodiehdotuksia samalla, kun ohjelmoija kirjoittaa koodia. Vaihtoehtoisesti kirjoittamalla koodiin kommentin siitä, mitä haluaisi ohjelmoida, Github Copilot voi generoida kommentin perustella koodia. Tämä voidaan nähdä kuvassa 1. (Github 2022.)

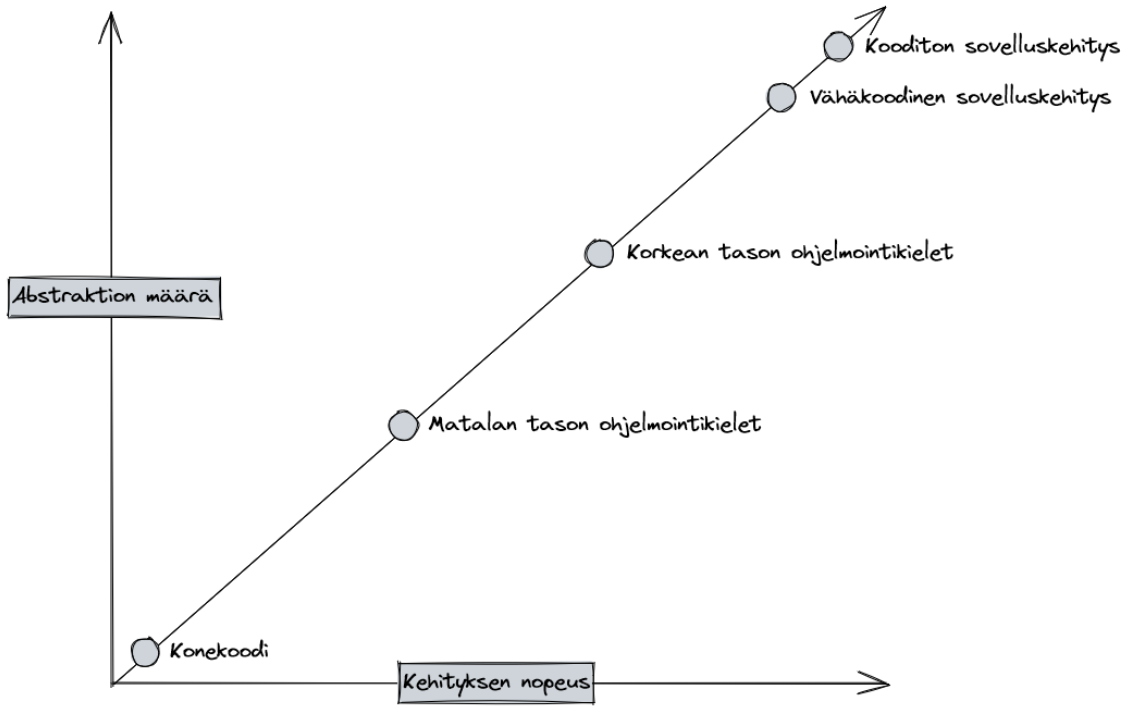


Kuva 1. Github Copilot generoi koodia funktion yläpuolella olevan kommentin perusteella. (Github 2022.)

Arkipäiväisempää koodigenerointia ovat koodieditorin tarjoamat toiminnallisuudet. Yksi esimerkki tällaisestä on luokkametodien generointi. Jotkin luokkametodit, kuten getterit ja setterit saattavat toistua koodissa satoja kertoja. Ne ovat kuitenkin lähes identtisiä toistensa kanssa. Suurimmat koodieditorit tarjoavatkin mahdollisuuden generoida luokalle sopivan getterin ja setterin. Täten säästyy ohjelmoijan aikaa.

Vähäkoodinen (engl. low-code) ja kooditon (engl. no-code) sovelluskehitys ovat myös yhdenlaista koodigenerointia. Vähäkoodisessa sovelluskehityksessä käytetään mahdollisimman vähän varsinaisen koodin kirjoittamista ja sovellukset luodaan käyttämällä visuaalisia käyttöliittymiä. Tällöin myös muutkin kuin ohjelmoijat voivat luoda ohjelmistoja. Kooditon sovelluskehitys on hyvin samankaltaista kuin vähäkoodinen sovelluskehitys. Vähäkoodinen sovelluskehitys vaatii jonkin verran ohjelmoijan apua manuaalisen koodauksen tai skriptauksen muodossa, kun taas kooditon sovelluskehitys on täysin itsenäinen menetelmä ja toimii pelkästään visuaalisten käyttöliittymien varassa. (IBM 2022.) Yksi esimerkki koodittomasta sovelluskehitysalustasta on Anima (Techcrunch 2021), johon

tässä opinnäytetyössä perehdytään. Ohjelmoinnin eri tasoja voidaan nähdä kuvassa 2. Kuvassa nähdään, kuinka abstraktion määrän noustessa myös kehitysnopeus kasvaa. Toisaalta abstraktion noustessa myös kontrolli koodista vähenee.



Kuva 2. Ohjelmoinnin eri tasoja.

2.3 Koodigeneroinnin hyödyt ja vaarat (sudenkuopat)

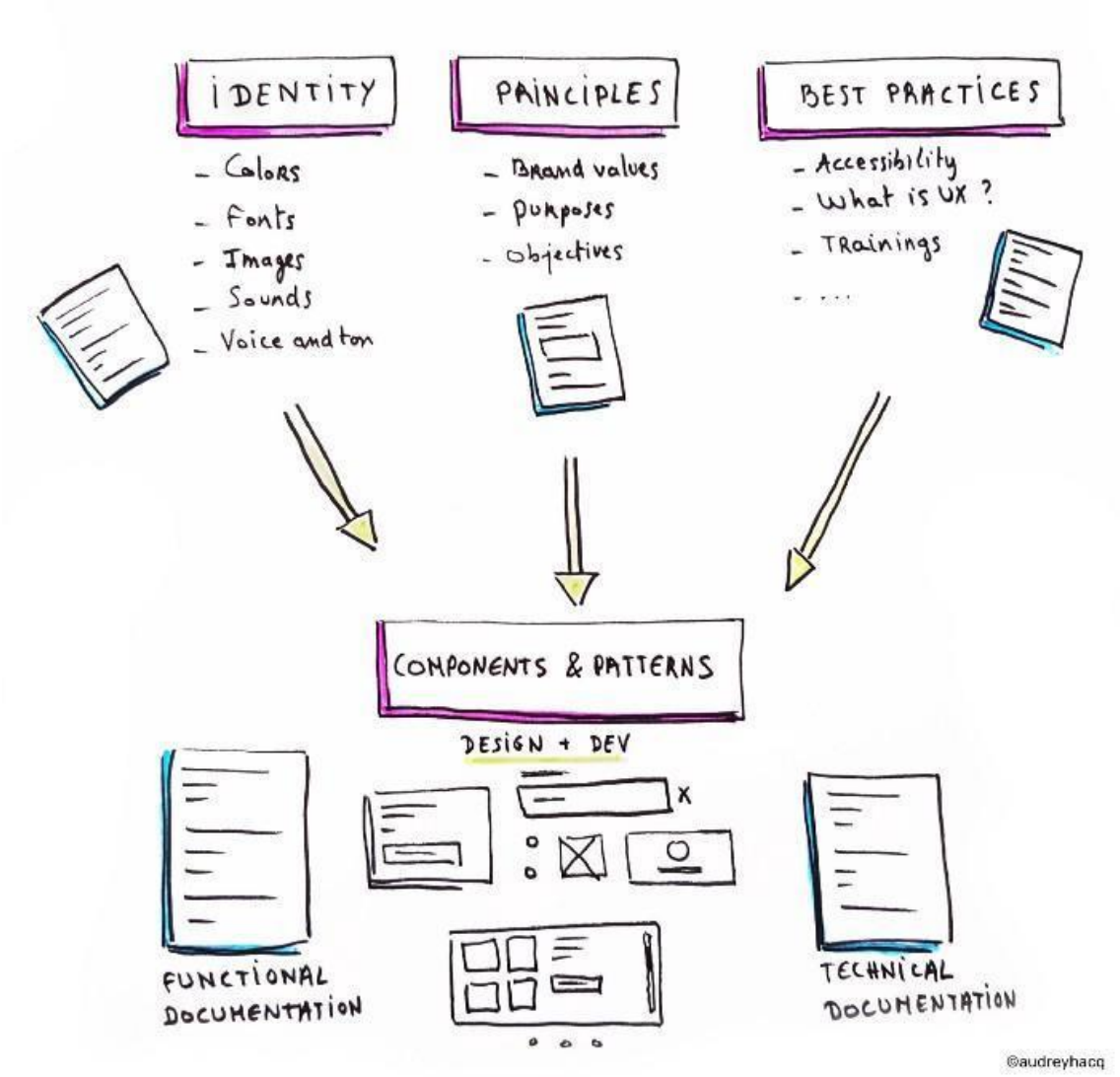
Kun abstraktion määrä nousee, ohjelmoijan suora kontrolli koodista vähenee. Tämä voi tuoda ongelmia, jos koodiin täytyy tehdä manuaalisia muutoksia. Siksi onkin tärkeää, että generoitu koodi on laadukasta, eli selkeää, yksiselitteistä, tehokasta ja virheetöntä. Esimerkiksi parhaimmillaan kooditon sovelluskehitys säästää aikaa ja rahaa, koska sovelluksen luominen kestää vähemmän aikaa sekä sen tekemiseen tarvitaan vähemmän asiantuntevaa henkilöstöä. Mutta siinä vaiheessa, kun sovellukseen pitäisi tehdä jokin ominaisuus, jota ei suoraan pysty käyttöliittymästä tekemään, jonkun täytyy tehdä se manuaalisesti. Tällaisia ominaisuuksia voi olla vaikea integroida korkean tason abstraktion koodiin, jos generoitu koodi ei ole laadukasta. Generoitua koodia ei myöskään

välttämättä ole tehty ihmisille luettavaksi, ja se saattaa olla sellaista, jota ihminen ei välttämättä itse kirjoittaisi.

3 Menetelmät

3.1 Suunnittelujärjestelmä

Suunnittelujärjestelmä on suosittu työkalu niin pienille kuin isoille yrityksille, joilla on monimuotoisia digitaalisia tuotteita. Pääidea suunnittelujärjestelmässä on tarjota yleisiä mutta räätälöitäviä rakennuspalikoita, joilla rakentaa käyttöliittymiä. Uudelleen käyttämällä rakennuspalikoita kehittäjät ja suunnittelijat voivat pitää käyttökokemuksen sekä käyttöliittymän yhdenmukaisena. (Izotov, 2020.) Suunnittelujärjestelmä toimii yksittäisenä totuuden lähteenä koko organisaatiolle (Adobe, 2022), jossa identiteetti, periaatteet ja parhaat käytännöt muodostuvat komponenteiksi ja malleiksi kuten kuvassa 3 nähdään.

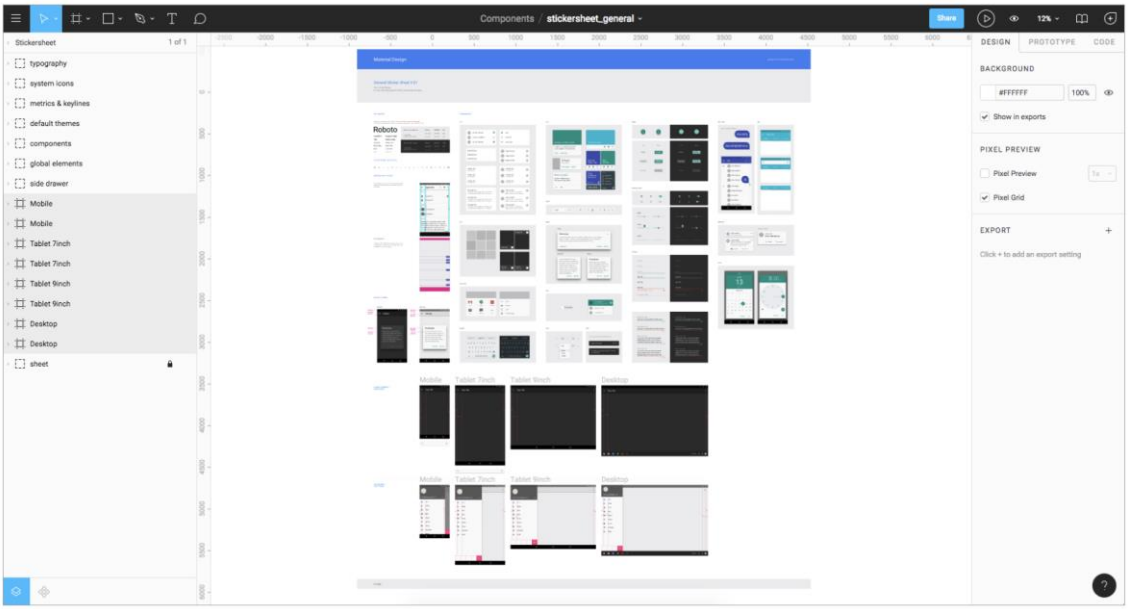


Kuva 3. Havainnollistava kuva suunnittelujärjestelmästä. (uxdesign, 2022.)

Suunnittelujärjestelmät ovat laajalti käytössä. Uxtoolsin kyselyn mukaan jo vuonna 2018 85 %:lla vastaajista suunnittelujärjestelmä oli jollain tasolla käytössä. Uxtools on tuotesuunnittelijoiden laaja yhteisö, joka tekee vuosittaisia kyselyitä liittyen käyttäjäkokemukseen. Edeltävänä vuonna uxtoolsin kyselyn prosentti oli vain 72. (Uxtools, 2018.) Vuoden 2021 kyselyssä 73 % vastasi, että he käyttävät Figmaa suunnittelujärjestelmän hallinnointiin (Uxtools, 2021).

Figma on suunnittelijan työkalu, joka yhdistää kaikki suunnitteluprosessissa olevat tahot, jotta he pystyvät toimittamaan parempia tuotteita nopeammin (Figma, 2022). Figmaa voi käyttää tietokoneella, selaimessa ja puhelimella. Se mahdol-

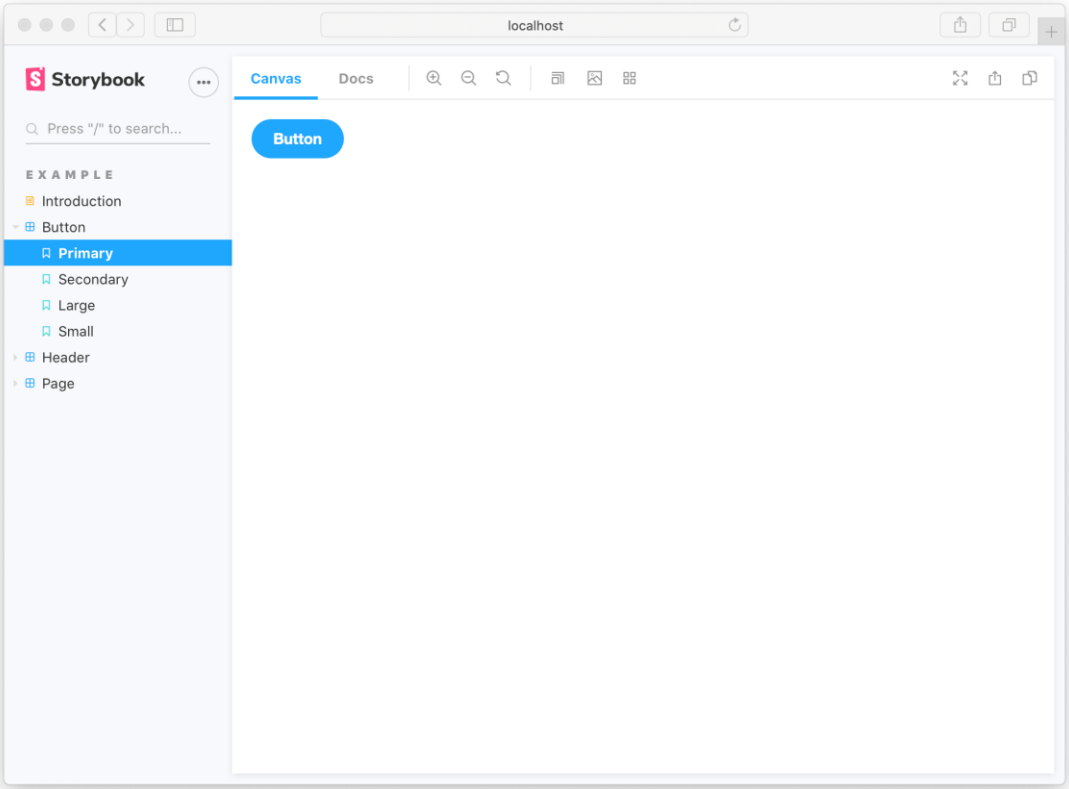
listaa reaaliaikaisen yhteistyön toisten suunnittelijoiden kanssa, jolloin sen käyttäjät voivat luoda käyttöliittymiä suuremmalla tehokkuudella. (Anima, 2022.) Figman käyttöliittymä nähdään kuvassa 4. Figmalla voidaan suunnitella suunnittelujärjestelmä, mutta sille myös tarvitaan koodissa vastine. Yksi suosittu työkalu siihen on Storybook.



Kuva 4. Figman käyttöliittymä (Toptal, 2022.)

3.2 Storybook

Storybook on kehittäjän työkalu kehittämään, testaamaan ja dokumentoimaan käyttöliittymän komponentteja (Storybook, 2022). Jokainen komponentti Storybookissa pitää sisällään oman tarinansa ja joissain komponenteissa tarinoita on useita. Tarina on webbisivu, jonka sisällä on komponentin käyttöohjeet, komponentin renderöinnin tulos ja sen dokumentaatio. Sivua voidaan käyttää esittelemään komponenttia muille, kuten kehittäjille, suunnittelijoille tai managereille. (Izotov, 2020.) Storybook tukee Reactia, Vuea, Angularia sekä web-komponentteja (Storybook, 2022). Storybookin käyttöliittymä nähdään kuvassa 5.



Kuva 5. Storybookin käyttöliittymä. Näemme “Button”-nimisen komponentin, johon on lisätty neljä eri tarinaa, “Primary”, “Secondary”, “Large” ja “Small”. “Canvas”-välilehdessä näemme komponentin renderöinnin tuloksen, ja “Docs”-välilehdessä olisi komponentin dokumentaatio. (Storybook, 2022.)

Komponentit, jotka toteutetaan suunnittelujärjestelmään, tulee olla puhtaita sekä esitteleviä (Storybook, 2022). Puhdas komponentti tuottaa aina saman lopputuloksen, jos sen tila tai parametrit eivät vaihdu (Chinda, 2022). Esittelevä komponentti ei tee muuta kuin renderöi itsensä (Accomazzo et al. 2017). Nämä komponentit käsittelevät käyttöliittymän ulkoasua, reagoivat vain niihin välitettyihin parametreihin, eivät sisällä sovelluskohtaista liiketoimintalogiikkaa, ja ovat agnostisia sen suhteen, miten dataa ladataan. Nämä kaikki ovat välttämättömiä, jotta komponentit ovat uudelleenkäytettäviä. (Storybook, 2022.)

3.3 React

React on Metan ylläpitämä ilmainen ja avoimen lähdekoodin komponenttipohjainen JavaScript-kirjasto, jolla rakennetaan käyttöliittymiä. Komponentit ovat omia

kapseloituja funktioita tai luokkia, jotka muodostavat monimutkaisia käyttöliittymiä. Reactissa komponentit kirjoitetaan JSX:ällä. JSX on JavaScriptin syntaksin laajennus, joka koodin kokoamisen jälkeen muuttuu normaaliksi JavaScript-koodiksi (React, 2022). Esimerkkikoodissa 1 on JSX:ällä määritelty React-elementti.

```
const element = <h1>Hello, world!</h1>
```

Esimerkkikoodi 1. JSX:ällä määritelty React-elementti.

Elementti määrittelee sen, mitä halutaan nähdä näytöllä. Mutta jotta elementit voidaan nähdä, ne pitää renderöidä selaimen Document Object Modeliin (DOM). Yleensä React-sovelluksissa HyperText Markup Languagella (HTML) määritellään yksi DOM-juurisolmu. (React, 2022.) Tämä nähdään esimerkkikoodissa 2.

```
<div id=”root”></div>
```

Esimerkkikoodi 2. DOM-juurisolmu HTML-tiedostossa.

Reactilla renderöinti tapahtuu ensin luomalla React DOM-juuri, johon välitetään juuri määritelty DOM-solmu. Tämän jälkeen React DOM:in juuresta voidaan kutsua metodia render. (React, 2022.) Tämä on esitelty esimerkkikoodissa 3.

```
const root = ReactDOM.createRoot(
      document.getElementById(“root”)
)
const element = <h1>Hello, world!</h1>
root.render(element)
```

Esimerkkikoodi 3. React DOM:in juurimuuttujan luonti ja sen käyttäminen renderöimään React-elementti.

Reactissa komponentit koostuvat elementeistä (React, 2022). Esimerkkikoodissa 4 on esimerkki Button-komponentista, ja esimerkkikoodissa 5 on esimerkki sen käytöstä. Kuvassa 6 nähdään lopputulos, jos Main-funktio renderöidään.

```
const Button = () => {
	return (
		<button>I am a button</button>
	)
}
```

Esimerkkikoodi 4. React-komponentti, jossa on Button-elementti.

```
const Main = () => {
	return (
		<div>
			<h1>Hello, world!</h1>
			<Button/>
		</div>
	)
}
```

Esimerkkikoodi 5. React-komponentti, joka käyttää Button-komponenttia.

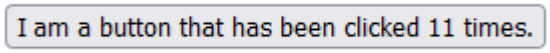
Kuva 6. Main-funktion renderöinnin lopputulos.

Reactin komponentteihin voidaan myös lisätä tila, jolloin komponentista tulee reaktiivinen. Esimerkiksi voimme lisätä Button-komponenttiimme tilan, joka pitää sisällään, kuinka monta kertaa nappia on painettu. Aina, kun komponentin tila muuttuu, React päivittää DOM:in. Esimerkkikoodissa 6 nähdään tila lisättynä Button komponenttiin, ja kuvassa 7 on napin renderöinnin tulos.

```
const Button = () => {
	const [count, setCount] = React.useState(0)
	return (
		<button
		onClick={() =>
			setCount(currentCount => currentCount + 1)
```

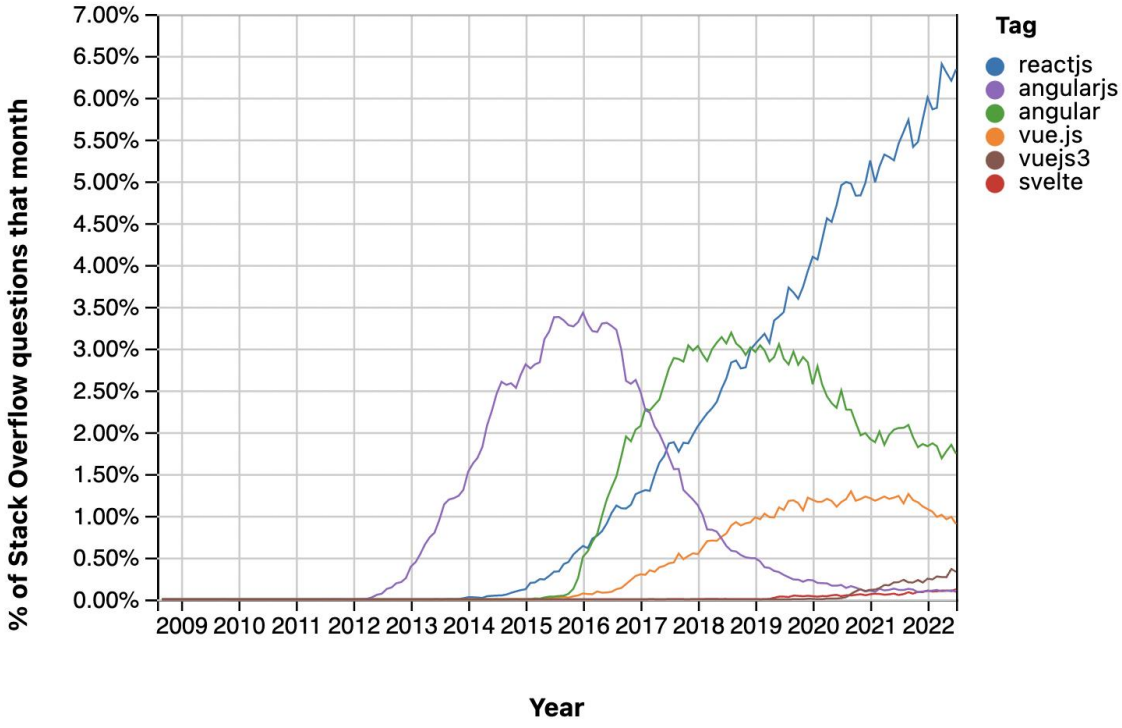
```
            }>
            I am a button that has been clicked {count} times.
            </button>
        )
}
```

Esimerkkikoodi 6. Komponenttiin lisätty tila “count”, ja nappiin lisätty tapahtumankäsittelijä, joka plussaa nykyisen tilan yhdellä.



Kuva 7. Uusi Button-komponentti, jota on klikattu 11 kertaa.

React on useiden eri kyselyiden mukaan suosituin frontend-kirjasto. Tämä voidaan huomata esimerkiksi Stackoverflow-postausten tagien perusteella, jotka nähdään kuvassa 8.



Kuva 8. Stackoverflow-postausten tagit yhden kuukauden aikana esitettyinä kuvaajassa.

Tyylit Reactissa

Reactin tyylittelyyn on useita eri menetelmiä. Kaikista yksinkertaisinta on käyttää Cascading Style Sheetsiä (CSS). Jos haluamme komponenttiimme CSS-luokkia, ne voidaan lisätä elementin parametrin “className” sisään (React, 2022.) Tämä nähdään esimerkkikoodissa 7.

```
<span className=”styled-text”>Text</span>
```

Esimerkkikoodi 7. Span-elementtiin lisätty CSS-luokka “styled-text”.

CSS:n sijaan myös Sass on suosittu vaihtoehto. Sass lisää ominaisuuksia CSS:ään, mutta kaikki CSS on myös validia Sass-koodia (Sass, 2022).

Kolmas vaihtoehto on CSS-in-JS, joka on malli, missä CSS rakennetaan käyttämällä JavaScriptiä. CSS-in-JS ei ole osa Reactia, mutta tulee osana kolmannen osapuolen kirjastoja (React, 2022.) Yksi esimerkki CSS-in-JS-kirjastosta on styled-components. Esimerkkikoodissa 8 nähdään styled-componentsilla tyylitelty Button-elementti, jonka taustaväri on punainen.

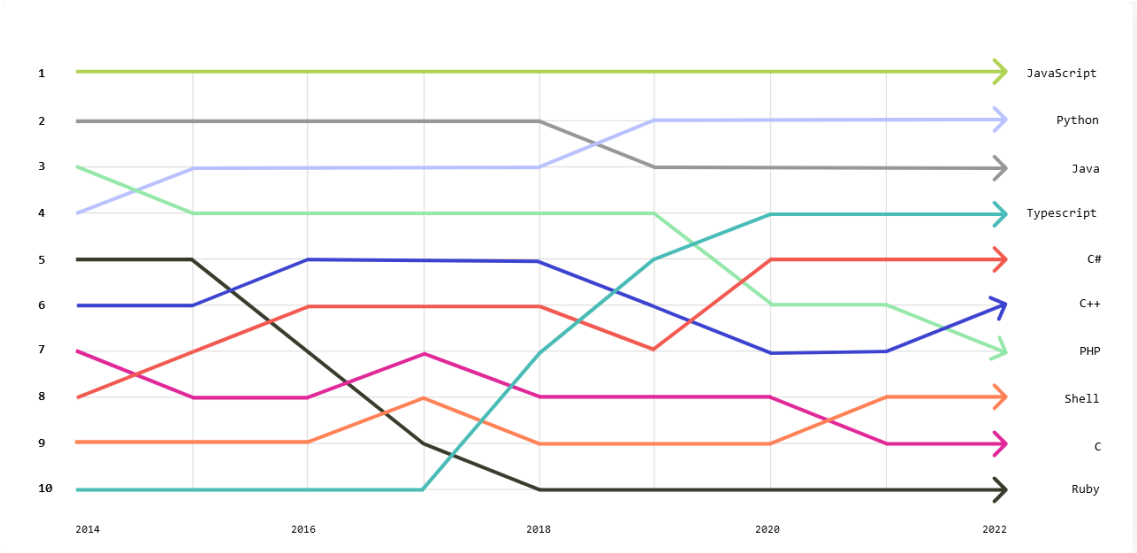
```
const Button = styled.button`
      background-color: red
`
```

Esimerkkikoodi 8. Styled-components-kirjastolla tyylitelty Button-elementti.

Reactissa on myös yleistä käyttää user interface (UI) -komponenttikirjastoja. UI-kirjasto sisältää käyttövalmiita komponentteja, kuten nappeja. Tällöin aloittelevan React-koodaajan ei tarvitse luoda elementtejä nollista, mikä säästää aikaa ja mahdollistaa nopeamman prototyyppauksen. Huonoja puolia on, että jos komponentteja ei kustomoi, sovellus näyttää samalta kuin kaikki muutkin samaa kirjastoa käyttävät sovellukset. Kirjastosta riippuen myös kustomointi voi olla haastavaa. (Varkki, 2021.) Yksi yleinen tapa on ottaa projektin pohjaksi jokin UI-kirjasto, kuten MUI, ja räätälöidä se omaan käyttöön sopivaksi. MUI perustuu Googlen Material Designiin.

3.4 TypeScript

React käyttää oletuksena JavaScriptiä, mutta myös tuki TypeScriptille löytyy. TypeScript on Microsoftin kehittämä JavaScriptiin pohjautuva kieli, joka koodia kääntäessä muuttuu JavaScriptiksi. (Yli-Hukkala, 2021.) Eli kaikki JavaScript-koodi on myös TypeScript-koodia**.** TypeScriptin päämääräinen tavoite on lisätä tyypit JavaScriptiin, sillä JavaScript on heikosti tyypitetty kieli. Heikosti tyypitetty koodikieli ei nimensä mukaan millään tapaa tarkasta muuttujien tyyppejä. Tämä voi olla ongelmallista, sillä yksi muuttuja voi yhdessä hetkessä olla merkkijono, toisessa numero, ja kolmannessa lista. TypeScript tuo koodiin rakennetta ja pakottaa koodin kirjoittajaa miettimään, miten muuttujia käytetään (Yli-Hukkala, 2021). TypeScriptin suosio on noussut räjähdysmäisesti viimeisen viiden vuoden aikana, mikä voidaan nähdä kuvasta 9.

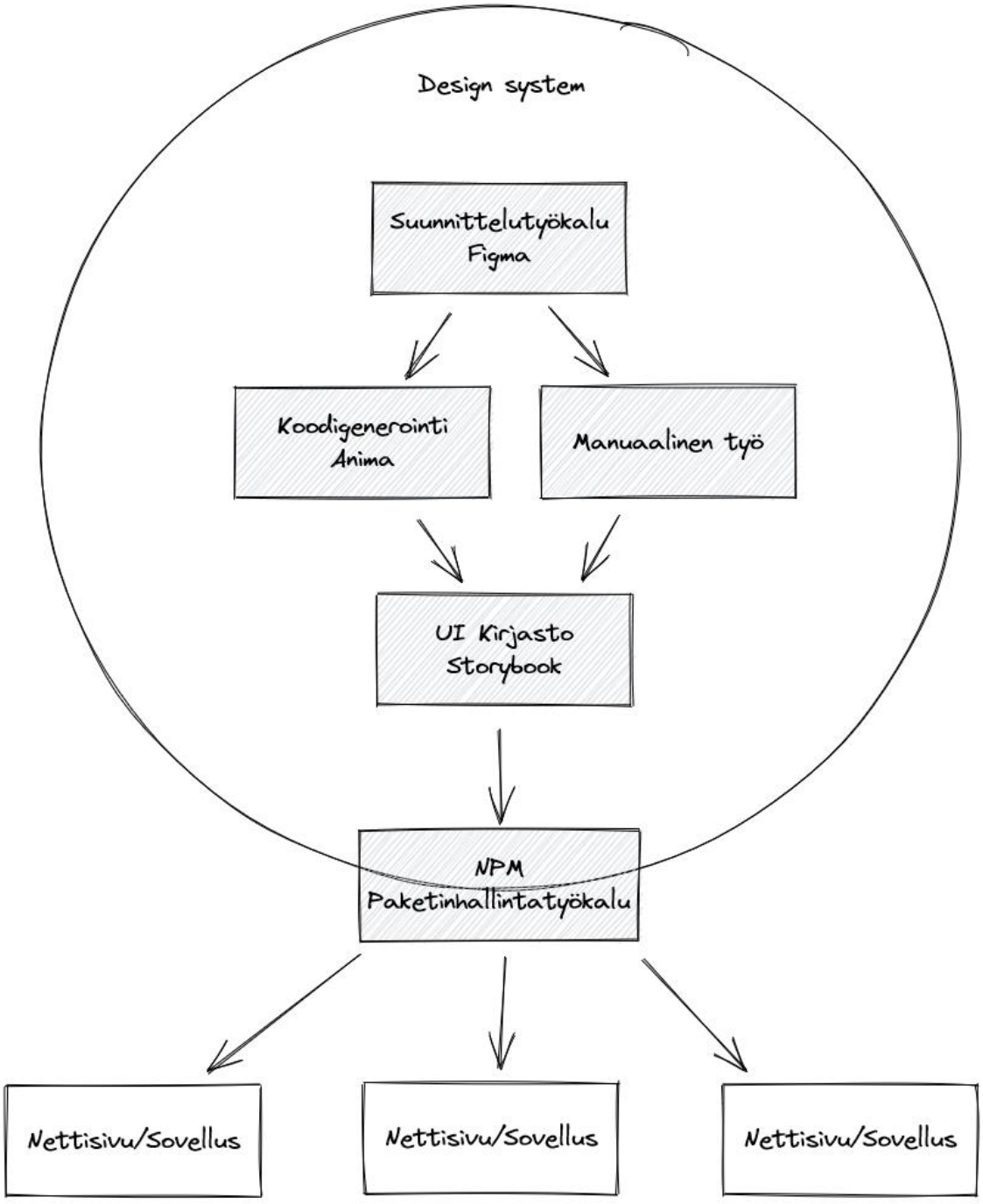


Kuva 9. TypeScript on GitHubissa 4. suosituin kieli vuonna 2022.

3.5 Koodigenerointi

Kun suunnitelmat ovat Figmassa ja suunnitelman toteutus tehty Reactilla Storybookiin, ne täytyy pitää jollain tavalla synkronissa. Niiden pitäminen synkronissa on kallis sekä loputon tehtävä, joka vaatii paljon ihmisen vaivannäköä. Tiimien pitää manuaalisesti huomata erot komponenteissa ja olla yhteisymmärryksessä siitä, mikä versio komponentista on oikea ja yliajaa toisen. (Anima, 2022.)

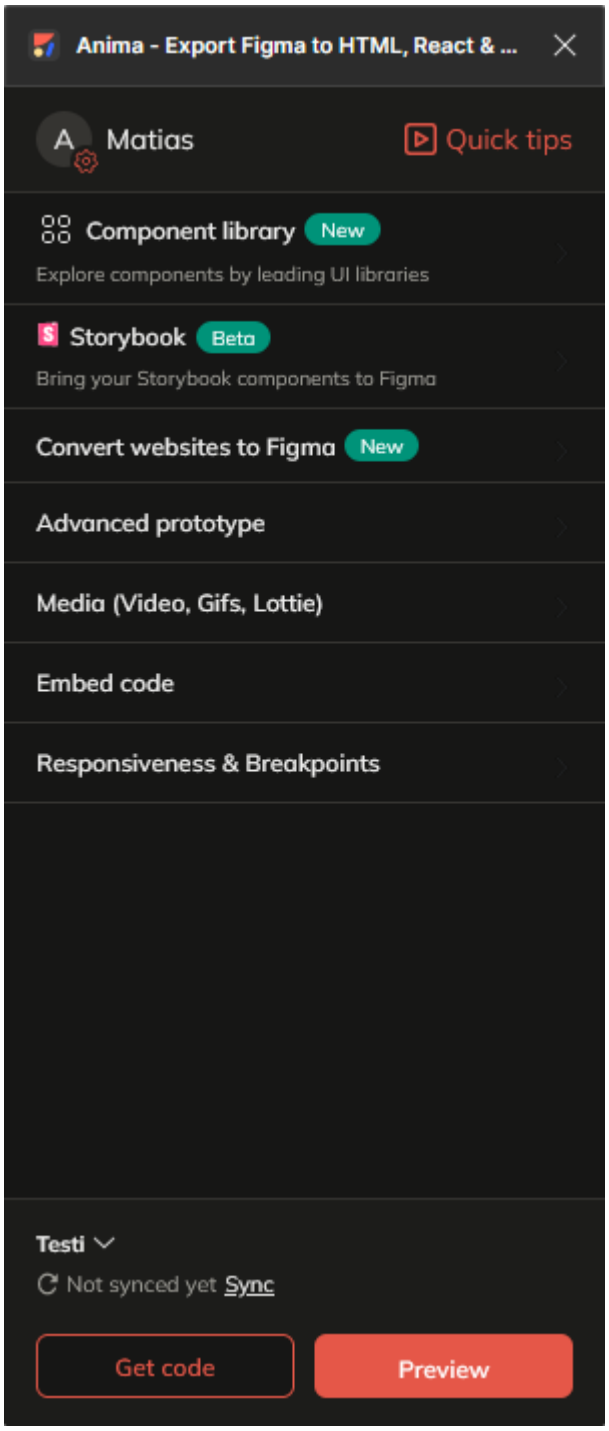
Tämä on ongelma, jossa voitaisiin hyödyntää koodigenerointia. Työn kulku suunnittelujärjestelmästä tuotantoon nähdään kuvassa 10.



Kuva 10. Työn kulku suunnittelujärjestelmää sekä koodigenerointia hyödyntäessä kahta mahdollista reittiä pitkin.

3.6 Figmasta koodiksi koodigeneroinnilla

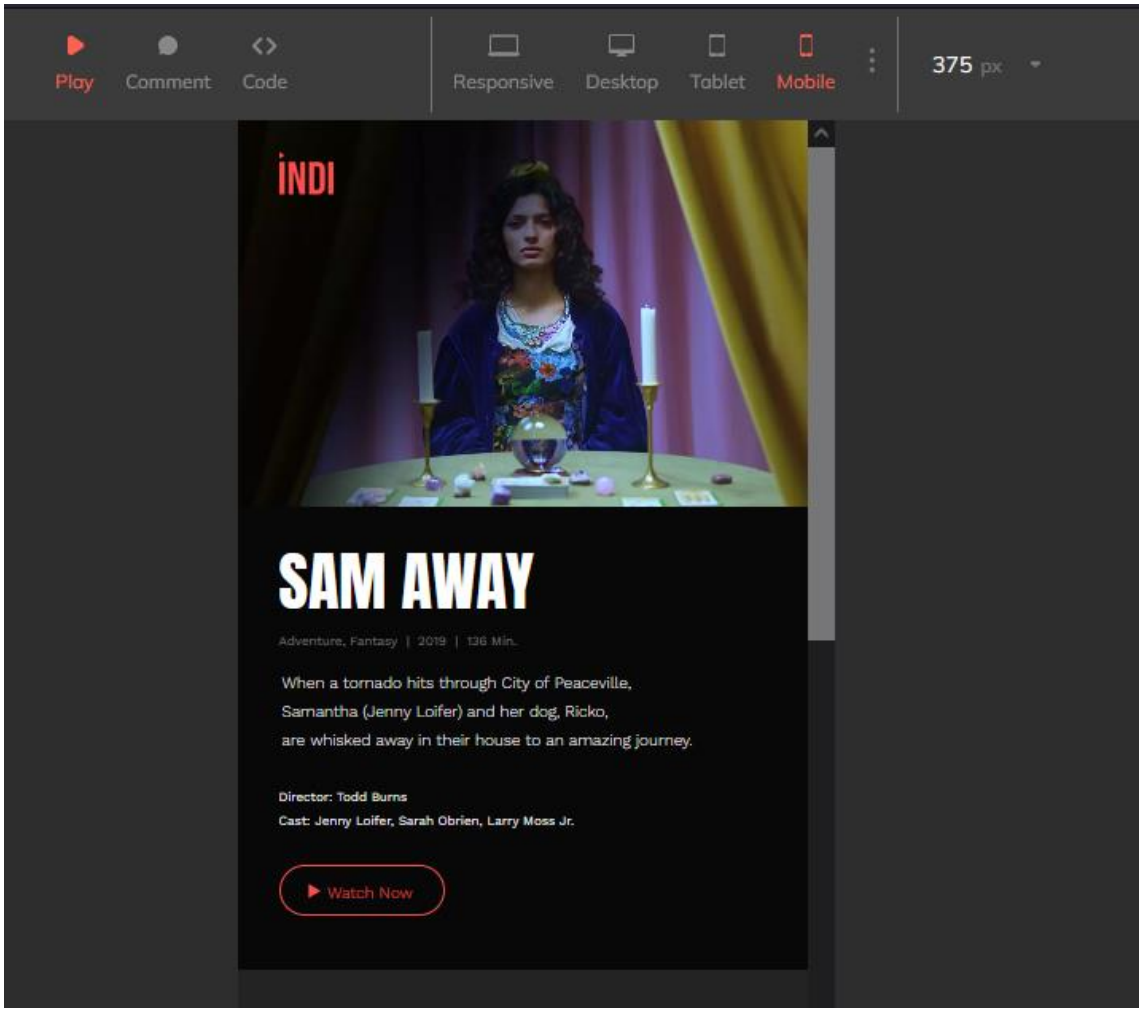
Figmaan on mahdollista asentaa lisäosia, jotka ovat yhteisön kehittämiä ohjelmia tai sovelluksia, joiden tarkoitus on laajentaa Figman ominaisuuksia. Hakusanalla “code generation” löytyy 58 eri koodigenerointiin liittyvää lisäosaa. Yksi näistä lisäosista on Anima, jolla pystyy generoimaan HTML-, CSS-, React- tai Vue-koodia ja rakentamaan koodiperäisiä prototyyppejä. Animan kehittäjät kertovat, että generoitu koodi on kehittäjäystävällistä monesta eri näkökulmasta: Anima tunnistaa automaattisesti toistuvat komponentit minimoiden koodin duplikaattiosat, Auto-Flexbox-niminen asettelu mahdollistaa dynaamisen sisällön suhteellisen asettelun. Lisäksi Animasta generoidulla koodilla ei ole ulkoisia koodiriippuvuuksilla mikä säästää kehittäjien aikaa antamalla heille kokonaisvaltaisen kontrollin koodista. (Figma, 2022.) Anima-lisäosan käyttöliittymä nähdään kuvassa 11.



Kuva 11. Figman Anima-lisäosan käyttöliittymä.

Ennen kuin pääsemme generoimaan koodia, Figman projekti täytyy synkronoida Animaan. Synkronoinnin jälkeen projektia voi esikatsella Animassa valitsemalla “Preview”. Esikatselu on Animan sivuilla sijaitseva prototyyppi. Jos pro-

jektiin on lisätty Anima-lisäosan mahdollistamia interaktiivisuuksia tai responsiivisuutta, nämä näkyvät prototyypissä. Figmasta viety näyttö Animan sivuilla nähdään kuvassa 12.

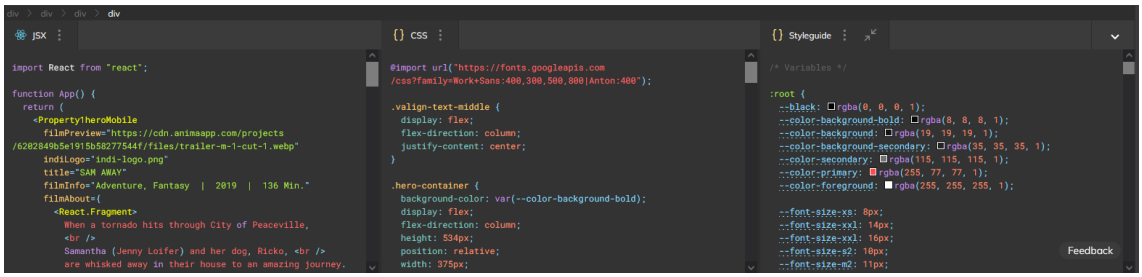


Kuva 12. Animan sivuille Figmasta viety näytön prototyyppi.

Valitsemalla “Get code” voimme valita kolmesta eri ohjelmistokehyksestä. Vaihtoehtoina ovat HTML, React ja Vue. Lisäosa kuvailee HTML:n olevan paras prototyypeille, käyttötestaukseen ja verkkosivuille. Reactin sekä Vuen kohdalla lisäosa kertoo, että voimme ladata joko koko React-sovelluksen tai yksittäisiä komponentteja. Tässä opinnäytetyössä haluamme React-koodia. Päästyämme Animan sivulle sivusto kysyy, haluammeko generoida koko projektin, yksittäisen Figma-näytön vai omavalinnaisen pätkän koodiksi. Jos valitsemme yksittäisen näytön, tai omavalinnaisen pätkän, avautuu myös vaihtoehto viedä koodi Code-

Sandboxiin. CodeSandbox on online-koodieditori ja prototyyppityökalu, joka nopeuttaa verkkosovellusten luomista ja jakamista. (CodeSandbox, 2022.) Ennen React-koodin generointia voimme vielä valita tyylittelyn sekä syntaksin. Tyylittelyn vaihtoehtoina on CSS, Styled Components ja Sass. Syntaksin voi valita funktionaalisten sekä luokkakomponenttien välillä.

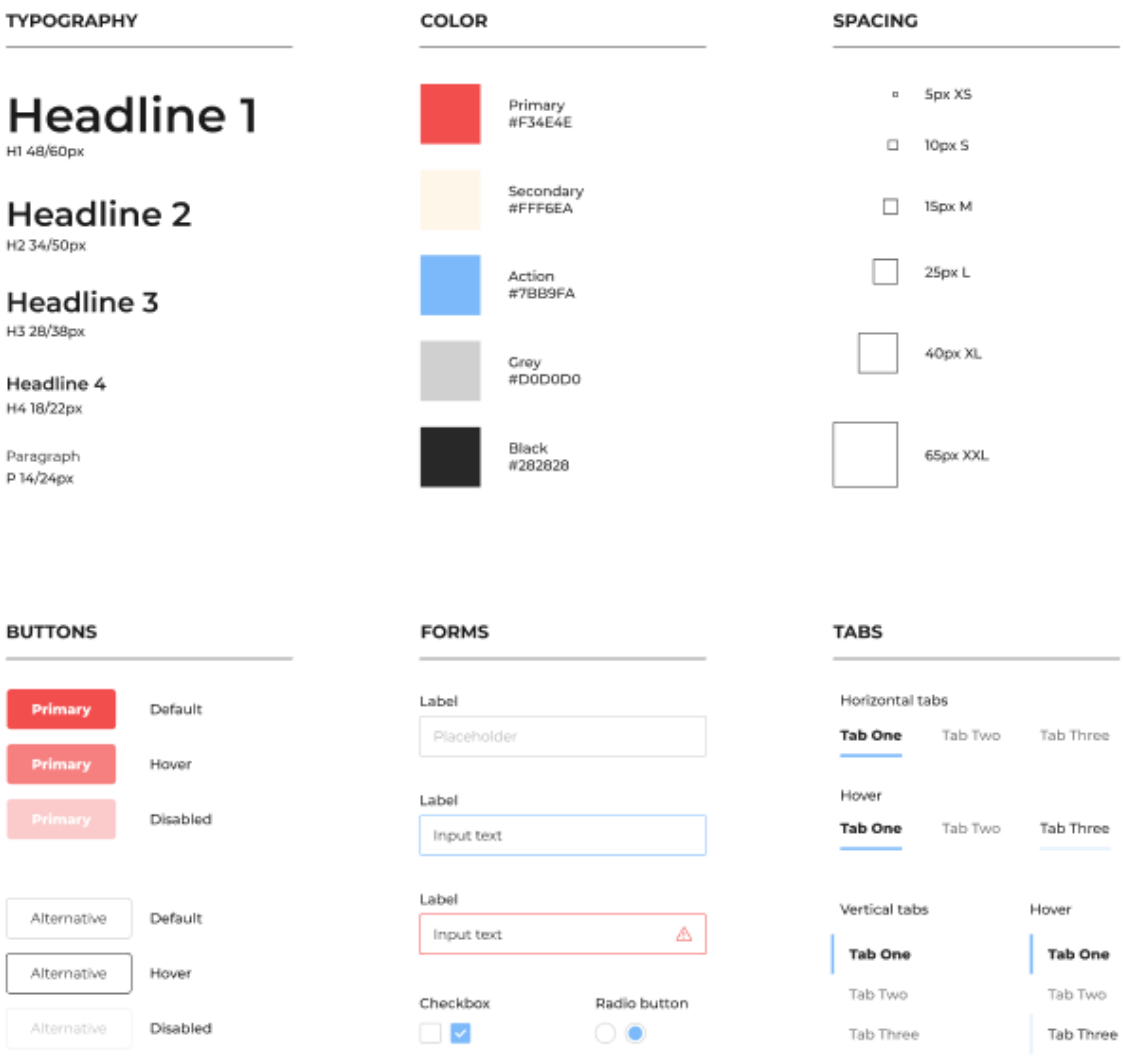
Omavalinnaisten elementtien ja komponenttien koodia voi myös tarkastella suoraan Animan käyttöliittymästä. Elementti valitaan klikkaamalla prototyypistä. Tällöin valinnan generoitu koodi tulee näkyviin sivun alareunaan, joka nähdään kuvassa 13. Vasemmalla on elementin React-koodi, keskellä elementin CSS, ja oikealla koko projektin tyyliohje, eli värit, fontit ja muut määriteltynä CSS:llä.



Kuva 13. Generoidun koodin katselmointia Animan käyttöliittymästä.

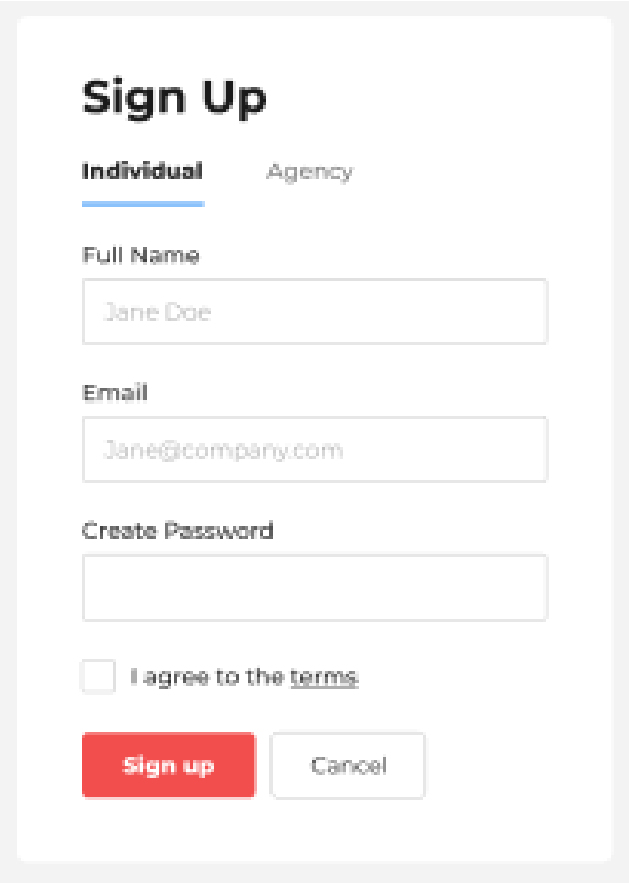
4 Toteutus

Figmassa otettiin käyttöön minimalistinen suunnittelujärjestelmä, joka sisältää typografian, värit, välitykset, napin, lomakkeen sekä välilehtinapin. Nämä kaikki nähdään kuvassa 14.



Kuva 14. Minimalistisen suunnittelujärjestelmän eri komponentit.

Näistä komponenteista voidaan koota lomake, joka nähdään kuvassa 15. Komponentit tullaan toteuttamaan Storybookiin, ensin manuaalisesti koodaamalla, ja sen jälkeen koodigeneroimalla käyttäen Animaa, jotta pystytään vertailemaan näiden kahden tavan eroja sekä tehokkuutta.



Kuva 15. Suunnittelujärjestelmän komponenteilla koottu lomake.

4.1 Figmasta Storybookiin manuaalisesti

Ensimmäisenä toteutetaan komponentit Storybookiin manuaalisesti koodaamalla.

4.1.1 Yleiset tyylit

Aloitetaan keräämällä suunnittelujärjestelmän typografia, värit ja välitykset yhteen CSS-tiedostoon. Ote tiedostosta nähdään esimerkkikoodissa 9.

```
@import url("https://fonts.googleapis.com/css?family=Montser-
rat:400,700,500,600,800");
```

```
:root {
	--black: #282828;
	--primary: #f34e4e;
	--secondary: #d0d0d0;
	--grey: #d0d0d0;
	--action: #7bb9fa;

	--font-size-m: 14px;
	--font-size-l: 18px;
	--font-size-xl: 28px;
	--font-size-xxl: 34px;
	--font-size-xxxl: 48px;

	--font-family-montserrat: "Montserrat";

	--spacing-xs: 5px;
	--spacing-s: 10px;
	--spacing-m: 15px;
	--spacing-l: 25px;
	--spacing-xl: 40px;
	--spacing-xxl: 65px;
}

.montserrat-extra-bold {
	font-family: var(--font-family-montserrat);
	font-weight: 800;
}

.montserrat-bold {
	font-family: var(--font-family-montserrat);
	font-weight: 700;
}

.montserrat-regular {
	font-family: var(--font-family-montserrat);
	font-weight: 500;
}
```

Esimerkkikoodi 9. Suunnittelujärjestelmän värit, välitykset sekä typografia yhdessä CSS-tiedostossa.

Tämän jälkeen otetaan käyttöön kyseinen CSS-tiedosto Reactin App.js-juuritiedostossa ja Storybookin preview.js-tiedostossa. Tällöin nämä CSS-muuttujat ovat saatavilla kaikkialla React- sekä Storybook-sovelluksissa.

4.1.2 Nappi

Luodaan seuraavat tiedostot komponenttia varten:

- button.jsx
- button.stories.jsx
- button.css
- index.jsx.

Ensimmäisenä toteutamme button.jsx-tiedoston. Tiedosto sisältää napin React-toteutuksen ja vie (engl. export) sen tiedostosta. Kun olio viedään tiedostosta, se asetetaan näkyville, jolloin se voidaan tuoda (engl. import) muihin tiedostoihin. Tiedoston sisältö nähdään esimerkkikoodeissa 10 ja 11. Nappiin lisätään PropTypet, jotka kuvaavat napin funktion parametrejä, kuten minkä tyyppinen napin parametri “primary” on. Proptypet ovat näkyvillä koodieditorissa sekä Storybookissa napin dokumentaatiossa. Dokumentaatio nähdään kuvassa 16.

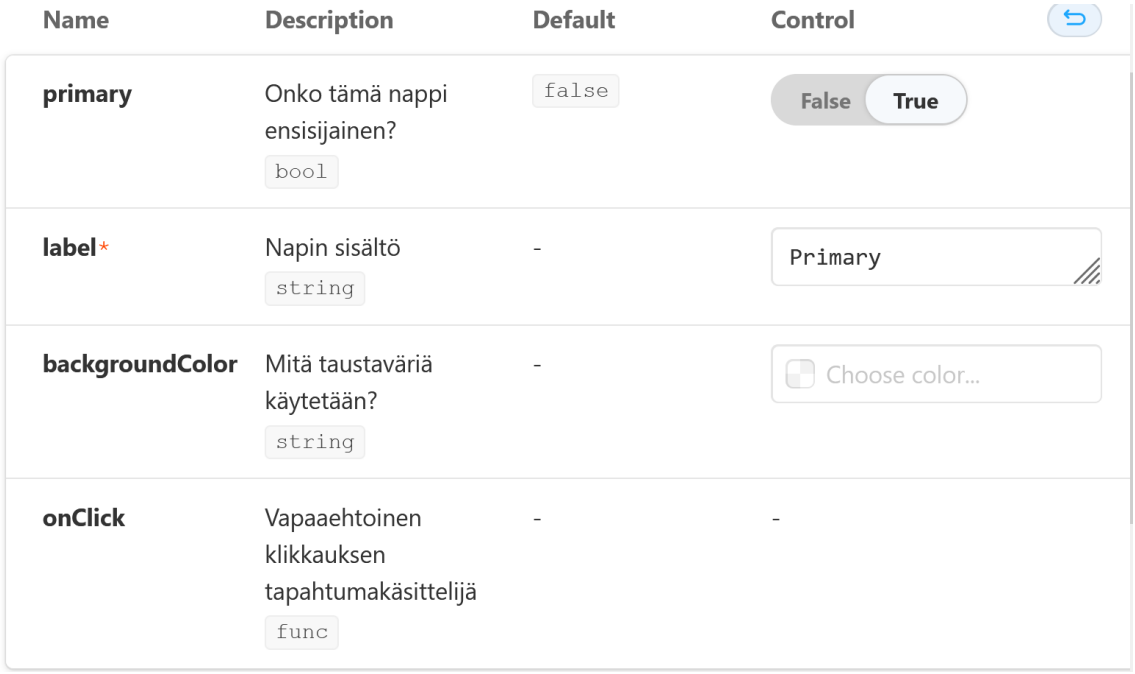
```
import "./button.css"

/**
 * Ensisijainen käyttöliittymäkomponentti käyttäjän vuorovaikutusta
varten
 */
export default function Button({
  primary = false,
  backgroundColor,
  label,
  ...props
}) {
  // Komponentin CSS-tyylit, jotka muodostuvat riippuen funktion para-
metreistä.
  const classes = [
```

```
        "storybook-button",
        "text-medium",
        primary
        ? "storybook-button--primary montserrat-extra-bold"
        : "storybook-button--secondary montserrat-light",
    ].join(" ")
    // Palautettava JSX koodi
    return (
        <button
            type='button'
            className={classes}
            style={backgroundColor && { backgroundColor }}
            {...props}
          >
            {label}
          </button>
    )
}
```

Esimerkkikoodi 10. Napin toteutus Reactilla.

```
Button.propTypes = {
  /**
   * Onko tämä nappi ensisijainen?
   */
  primary: PropTypes.bool,
  /**
   * Mitä taustaväriä käytetään?
   */
  backgroundColor: PropTypes.string,
  /**
   * Napin sisältö
   */
  label: PropTypes.string.isRequired,
  /**
   * Vapaaehtoinen klikkauksen tapahtumakäsittelijä
   */
  onClick: PropTypes.func,
}
```

Esimerkkikoodi 11. Napin PropTypet, jotka kuvaavat napin funktion parametrejä. Kommentit kuvaavat, mitä parametrit tekevät, ja ne ovat näkyvillä Storybookissa sekä koodieditorissa.



Kuva 16. Napin PropTypet näkyvillä napin dokumentaatiossa Storybookissa.

Seuraavaksi toteutamme button.stories.jsx-tiedoston. Se sisältää napin Storybook-konfiguraation. Tiedoston sisältö nähdään esimerkkikoodissa 12.

```
import React from "react"

import Button from "./Button"

export default {
      // Komponentti otsikkotasolla
      title: "Opinnäytetyö/Manuaalinen/Itsenäinen/Button",
      // Käytettävä komponentti
      component: Button,
      // Storybookissa komponentille voidaan antaa erilaisia
argumentteja, komponentin
      // parametreihin. Napin parametrille "backgroundColor", voidaan
antaa
      // kontrolleri "color". Tällöin käyttäjä Storybookissa pystyy
muuttamaan napin
      // taustaväriä käyttämällä interaktiivista väripalettia.
```

```
      argTypes: {
            backgroundColor: { control: "color" },
      },
}

// Napista luodaan pohja
const Template = (args) => <Button {...args} />

// Pohja "bind"ataan, jonka jälkeen sen "args"iin voidaan lisätä eri-
laisia parametrejä.
export const Primary = Template.bind({})
// Ensisijainen nappi
Primary.args = {
      primary: true,
      label: "Primary",
}

export const Secondary = Template.bind({})

// Toissijainen nappi
Secondary.args = {
      primary: false,
      label: "Alternative",
}
```

Esimerkkikoodi 12. Napin Storybook-konfiguraatio.

Seuraavana on button.css-tiedosto, joka sisältää napin tyylittelyn. Tiedoston sisältö nähdään esimerkkikoodissa 13.

```
.storybook-button {
      border: 0;
      border-radius: 4px;
      cursor: pointer;
      display: inline-block;
      padding: 12px 26px;
}

.storybook-button--primary {
      color: white;
      /* Voimme käyttää aiemmin määriteltyjä
```

```
        suunnittelujärjestelmän värejä hyödyntämällä
        css-muuttujia.*/
        background-color: var(--primary);
}

.storybook-button--secondary {
        color: var(--black);
        background-color: transparent;
        border: 1px solid var(--secondary);
}
```

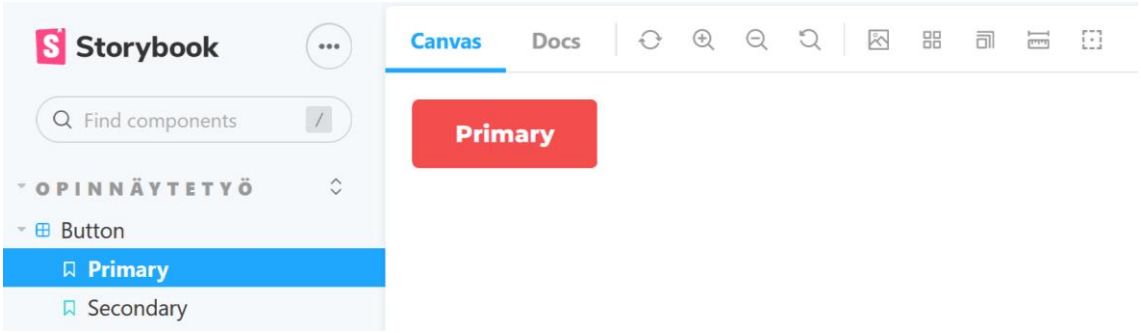
Esimerkkikoodi 13. Napin CSS-tiedosto, jossa on määritelty CSS-luokat.

Viimeisenä on index.jsx-tiedosto. Sen ainoa tavoite on uudelleenviedä nappi, jolloin se voidaan tuoda napin juurikansiosta. Tiedoston sisältö nähdään esimerkkikoodissa 14.

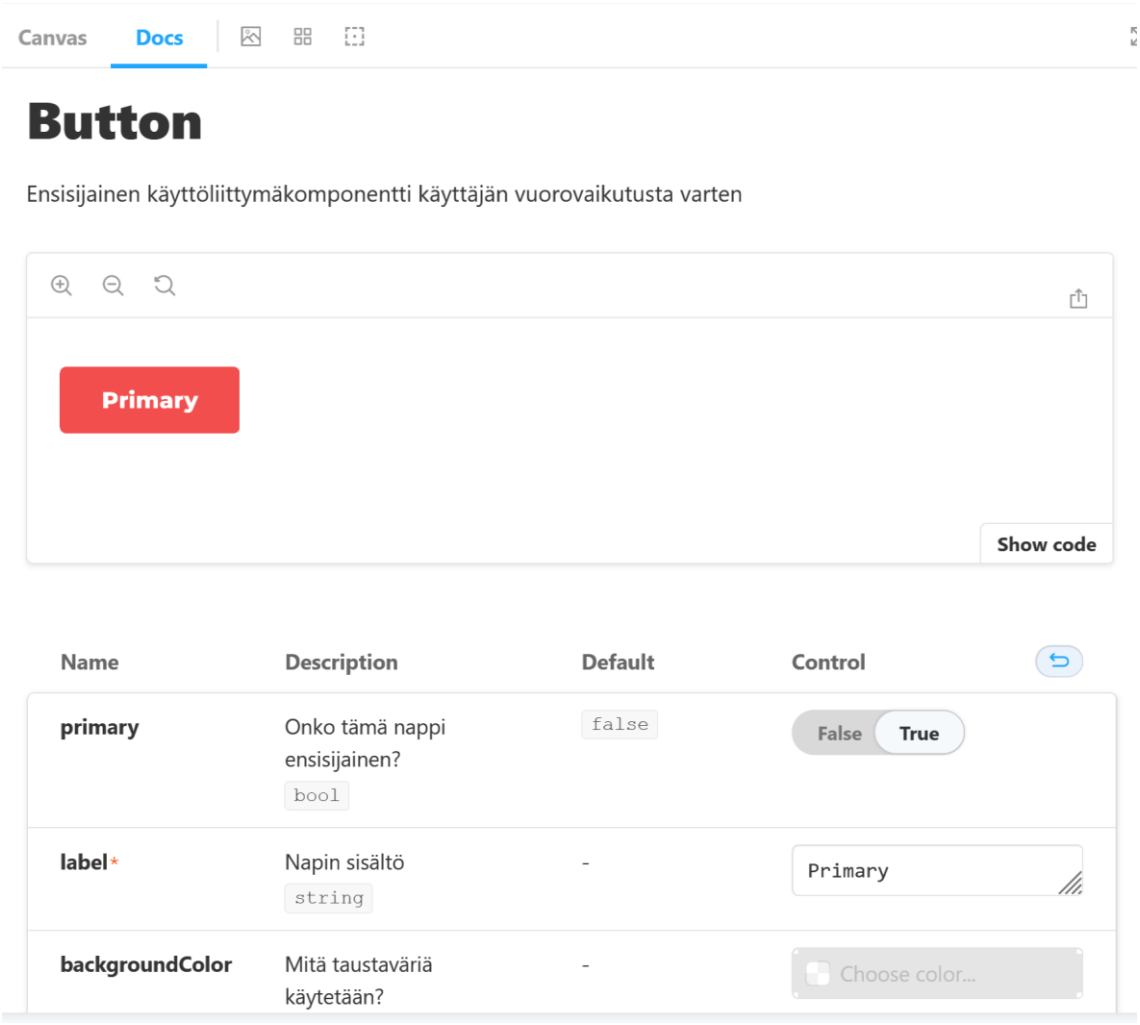
```
export { default } from "./Button"
```

Esimerkkikoodi 14. Nappi viedään uudelleen sen toteutustiedostosta. Hyöty tässä on se, että komponentti voidaan tuoda käyttämällä “import Button from ‘src/components/Button’”. Ilman uudelleen vientiä, tiedoston tuonti tapahtuisi “import Button from ‘src/components/Button/Button’”.

Lopputuloksena saamme napin, jonka käyttöliittymä vastaa suunnittelujärjestelmää. Nappia voidaan käyttää React-sovelluksessa. Storybookista löytyy napin esikatselu sekä dokumentaatio. Nappi Storybookissa nähdään kuvassa 17 ja sen dokumentaatio kuvassa 18.



Kuva 17. Napin esikatselu Storybookissa.



Kuva 18. Napin dokumentaatio Storybookissa.

4.1.3 Muut komponentit

Suunnittelujärjestelmän muut komponentit toteutettiin samalla tavalla kuin nappi toteutettiin. Selkeyden vuoksi Storybookissa komponentit jaetaan kansioihin sen perusteella, onko komponentti itsenäinen vai koostettu. Tämä voidaan tehdä määrittelemällä komponentit stories-tiedoston viennin olion title-attribuuttiin haluttu polku. Esimerkkikoodissa 14 nähdään polku itsenäiselle nappi-komponentille.

```
title: "Opinnäytetyö/Itsenäinen/Button",
```

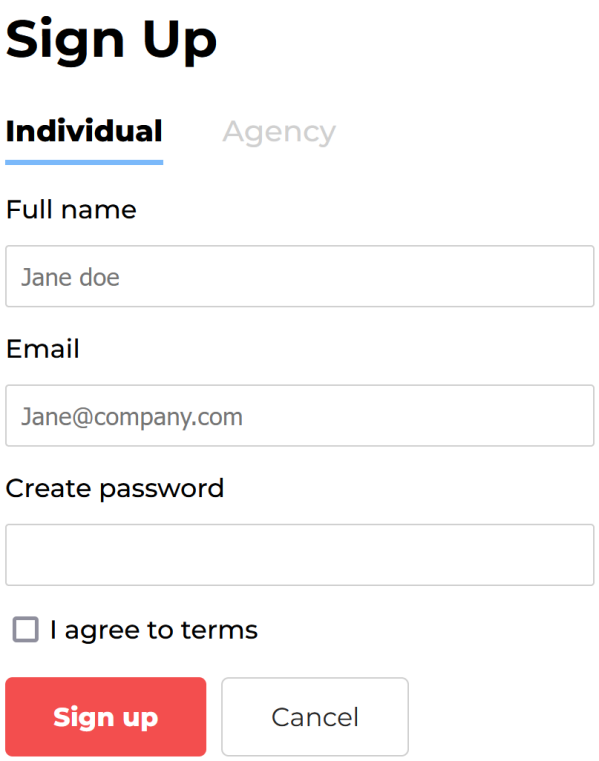
Esimerkkikoodi 14. Polku määrittelee napin sijainnin Storybookissa. Tällä määritelmällä nappi sijaitsee Itsenäinen-kansiossa, joka taas on opinnäytetyöprojektin sisällä.

4.1.4 Koostettu komponentti

Seuraavaksi koostettiin lomake aiemmin luoduista komponenteista. Ote Form-komponentista nähdään esimerkkikoodissa 15. Kuvassa nähdään, kuinka tiedostoon tuodaan tarvittavat komponentit, jonka jälkeen niitä käytetään osana Form-komponenttia. Form-komponentin renderöinnin tulos nähdään kuvassa 19.

```
import Button from "../Button"
import Checkbox from "../Checkbox"
import Input from "../Input"
import Tabs from "../Tabs"
import "./form.css"

export default function Form({ title = "Sign Up" }) {
  const [selected, setSelected] = React.useState(0)
  return (
      <div className="storybook-form-container">
        <h1 className='montserrat-bold text-xl'>{title}</h1>
        <Tabs
          tabs={["Individual", "Agency"]}
          selected={selected}
          setSelected={setSelected}
          className="storybook-form-tabs"
        />
        <Input
          label='Full name'
          placeholder='Jane doe'
          className="storybook-form-input"
        />
        <Input
          label='Email'
          placeholder='Jane@company.com'
          className="storybook-form-input"
        />
```

Esimerkkikoodi 15. Tiedostoon tuodaan aiemmin luodut komponentit, jonka jälkeen niitä voidaan käyttää lomake-komponentin sisällä.



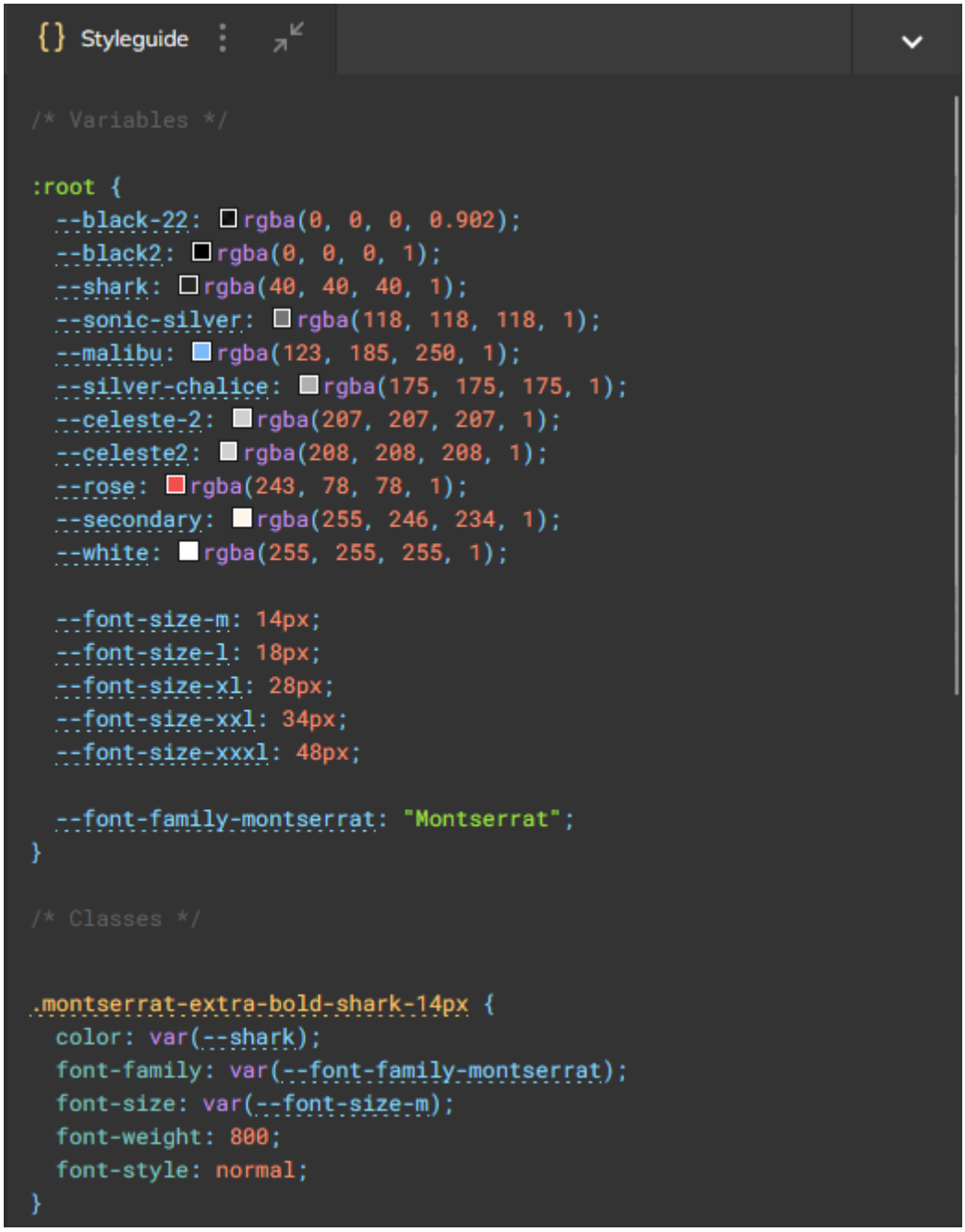
Kuva 19. Lomake-komponentin renderöinnin tulos.

4.2 Figmasta Storybookiin koodigeneroimalla

Seuraavaksi toteutetaan samat komponentit Storybookiin hyödyntämällä Animan koodigenerointia.

4.2.1 Yleiset tyylit

Viemällä Figman näytön Animaan saamme siinä käytetyt yleiset tyylit. Ote tyyliohjetiedostosta nähdään kuvassa 20. Kuvasta huomaamme, että Anima samankaltaisesti kuin manuaalisessa osiossa ryhmittää globaalit CSS-muuttujat :rootin alle. Anima ei kuitenkaan nimeä värien muuttujia sen mukaan, miten ne on nimetty Figmassa, vaan sen sijaan nimeää ne itse värin mukaan. Animan generoima tyyliohje myös tarjoaa typografialuokkia helpottaen tekstin tyylittelyä koodissa.



Kuva 20. Animan generoimat suunnittelujärjestelmän tyylit.

4.2.2 Nappi

Valitsemalla ensisijaisen napin Animassa saamme sen koodin. JSX- sekä CSS-koodi nähdään kuvassa 21. Kuvasta huomaamme, että CSS-tyyleissä on käytössä aiemmin määritellyt CSS-muuttujat. JSX-koodi on yksinkertainen. Komponentti ottaa vastaan vain sen sisälle määriteltävän sisällön. Huomaamme myös,

että nappi käyttää div-elementtejä. Animalla ei ole kontekstia siitä, millainen komponentti on, eikä sitä pysty erikseen määrittelemään. Joten Anima aina generoi elementit div-elementteinä. Figmassa Animan lisäosalla on kuitenkin mahdollista käyttää ulkoisia komponenttikirjastoja, kuten aiemmin mainittua mui:ta (Anima 2022). Tässä esimerkissä sitä ei kuitenkaan käytetty.

JSX

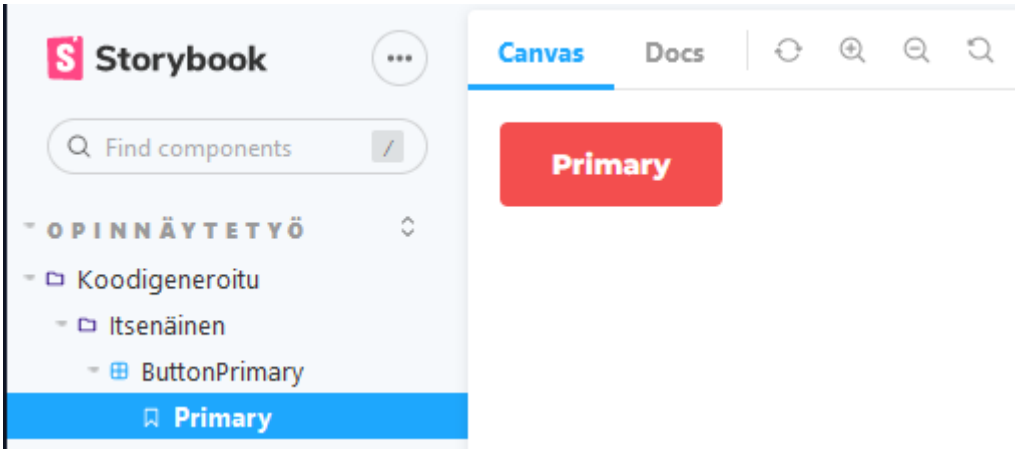
```jsx
import React from "react";

function App() {
  return <ButtonPrimaryHover2>Primary</ButtonPrimaryHover2>;
}

export default App;

function ButtonPrimaryHover2(props) {
  const { children } = props;

  return (
    <div className="button-primary-default">
      <div className="primary montserrat-extra-bold-white-14px">
        {children}
      </div>
    </div>
  );
}
```

CSS

```css
@import url("https://fonts.googleapis.com
/css?family=Montserrat:400,700,500,600,800");

.button-primary-default {
  align-items: flex-start;
  background-color: var(--rose);
  border-radius: 4px;
  display: flex;
  height: 40px;
  min-width: 109px;
  padding: 10px 0;
}

.primary {
  letter-spacing: 0;
  line-height: normal;
  text-align: center;
  width: 109px;
}
```

Kuva 21. Ensisijainen nappi koodigeneroituna. Anima ei aina saa Figmasta nimeä oikein, komponentin nimen pitäisi olla ButtonPrimary.

Lisäsimme Storybookiin koodigeneroinnille oman kansion, johon lisättiin koodigeneroitu nappi. Nappiin jouduttiin tekemään pieniä tyylittelyn muutoksia. Koodigeneroitu nappi nähdään kuvassa 22.



Kuva 22. Animalla koodigeneroitu nappi.

Animan mukaan komponenteista on mahdollista myös generoida variaatioita, mutta tätä ei saatu toimimaan. (Anima 2022.) Tämän avulla generoitu komponentti olisi enemmän vastannut manuaalisesti luotua, jossa yksi komponentti tarjoaa eri variaatioita. Ilman tätä jokainen napin variaatio on oma komponenttinsa. Napin eri tilat, kuten hover, piti myös lisätä manuaalisesti sekä myös nappi vaihdettiin käyttämään HTML-elementtiä button divin sijaan.

4.2.3 Muut komponentit sekä koostettu komponentti

Muiden komponenttien lisääminen tapahtui samalla tavalla kuin napinkin. Jokaisen komponentin kohdalla täytyi tehdä jonkin verran manuaalista säätöä.

Koostettu komponentti rakennettiin samalla tavalla kuin manuaalisessa metodissa. Animalla on myös mahdollista suoraan generoida suurempiakin valintoja. Tällöin pitää kuitenkin tarkoin määritellä sekä nimetä Figmassa olevat suunnitelmat, sillä se vaikuttaa suuresti generoidun koodin laatuun.

5 Tulokset ja pohdinta

Lopputuloksena saatiin käyttövalmiita React-komponentteja, jotka ulkonäöltään vastaavat Figmassa olevia käyttöliittymäsuunnitelmia. Komponentit lisättiin Storybookiin, jossa voidaan tarkastella niiden dokumentaatiota, esikatsella komponenttia sekä testata toiminnallisuuksia. Komponentit jaettiin eri kategorioihin niin hakemistossa kuin Storybookissa, jotta tulosten tarkastelu olisi helpompaa.

Animalla generoitiin React-komponentteja Figmassa olevien käyttöliittymäsuunnitelmien pohjalta. Jotta Anima toimisi hyvin, pitää Figmassa osata luoda ja nimetä komponentti tietyllä tavalla. Komponentin luomisesta ja nimeämisestä löytyy kuitenkin puutteellisesti Animan dokumentaatiota, mikä hidasti työtä. Esimerkiksi tästä johtuen suunnittelija joutuu näkemään enemmän vaivaa saadakseen hyviä tuloksia, eikä Anima sen takia välttämättä sovellu projekteihin, joissa on tiukka aikataulu tai budjetti, eikä ylimääräisille kokeiluille ole aikaa.

Animalla generoidessa Figman näytöstä saadaan siinä käytetyt yleiset tyylit, jotka Anima kokoaa yhteen teematiedostoon. Tämänkaltainen tyylien kokoaminen on hyvä käytäntö, koska tyylejä on helppo käyttää sekä myöhemmin muuttaa, sillä ne sijaitsevat vain yhdessä paikassa.

Anima myös tarjoaa mahdollisuuden generoida tyylit joko CSS:llä tai SCSS:llä, jolloin esimerkiksi tyylejä olisi myös helppo käyttää jo valmiissa projektissa. Koska teematiedosto saadaan generoitua Animaan yhdellä klikkauksella Figmasta, on iteraatio hyvin nopeaa. Generoidut React-komponentit ovat yksinkertaisia, eikä niihin pysty erikseen lisäämään toiminnallisuuksia. Toisaalta komponenttien yksinkertaisuus myös johtaa siihen, että koodigenerointi ei lisää projektiin abstraktia koodia. Jos komponentissa on tekstiä, komponentti ottaa sen vastaan funktion parametrinä komponenttipuussa korkeammalla sijaitsevalta komponentilta. Tämä voi saada komponentit näyttämään sekavilta, jos yritetään generoida liian isoja valintoja.

Kuten aiemmin myös on mainittu, komponenteista on mahdollista generoida myös variaatioita. Tätä ei kuitenkaan saatu toimimaan dokumentaation puutteellisuuden vuoksi. Ilman tätä toiminnallisuutta Anima generoi jokaisen variaation erikseen. Komponentit ulkonäöllisesti vastasivat suunnitelmaa lähes täydellisesti, mutta jokaisen komponentin kohdalla piti tehdä manuaalista korjausta. Manuaalisen työn määrä riippuu siitä, miten hyvin Figmassa komponentit on suunniteltu Animaa varten.

Huomioitavaa koodigeneroidessa on myös se, että Animalla ei ole Figman pohjalta mitään kontekstia siitä, missä elementtejä käytetään. Tämä tarkoittaa sitä, että Anima ei tunnista HTML-elementtejä, jolloin ne täytyy itse tyylitellä sekä ottaa käyttöön koodissa.

Koodigenerointi on vain yksi Animan ominaisuuksista. Anima markkinoi sivuillaan eniten sitä, kuinka se mahdollistaa interaktiivisten sekä responsiivisten prototyyppien luonnin. Näitä toiminnallisuuksia ei kuitenkaan ehditty testaamaan,

eikä sen myötä niiden vaikutuksia pystytty todentamaan esimerkiksi koodigeneroinnissa. Anima on myös hiljattain julkaissut mahdollisuuden synkronoida React-komponentteja Figmaan. Tällöin koodi olisi suunnittelujärjestelmän yksittäinen lähde.

Eniten hyötyä Animan koodigeneroinnista saa, kun sitä hyödynnetään uusissa projekteissa. Koodigeneroinnista voi olla myös hyötyä vanhoissa projekteissa generoimaan uusia komponentteja. Vanhoissa projekteissa on kuitenkin yleensä käytetty jotain tiettyä konventiota luomaan komponentteja, joista koodigenerointi todennäköisimmin eroaa. Uusissa projekteissa koodigeneroinnin hyötynä on se, että koodigenerointi voi olla se konventio, jota käytetään. Tällöin Anima mahdollistaa hyvin nopean kehityksen sekä iteraation komponenttien ulkonäön kannalta. Koodigeneroinnin todelliset hyödyt kuitenkin nähtäisiin pitkässä juoksussa, kun on jo olemassa suurempi komponenttikirjasto, johon tehdään paljon muutoksia. Pienemmissä projekteissa koodigeneroinnin sekä manuaalisen koodauksen ero ei välttämättä ole niin iso, että koodigenerointiin olisi kannattava panostaa. Koodigenerointi voi näissäkin tapauksissa nopeuttaa työtä, mikäli kehittäjä ei ole erityisen kokenut.

6 Kehitysmahdollisuudet ja tulevaisuudennäkymät

Koska nykyiset suuret web-kehykset ovat vielä suhteellisen uusia, niin ovat myös niiden koodigeneraattorit. Web-maailma on myös jatkuvasti kehittyvä, ja myös React on vastikään vakiintunut suosituimmaksi kirjastoksi web-sovellusten luomiselle. React on samaan aikaan myös alati muuttuva kirjasto. Esimerkiksi vuonna 2019 siihen julkaistiin päivitys, joka mahdollisti funktionaalisten komponenttien luonnin, mikä onkin nykypäivän standardi. Tällaisessa maailmassa jokin koodigeneraattori voi hyvin nopeasti vanhentua ja käydä täysin turhaksi. Tämä saattaa myös olla syy sille, miksi mikään suuri yritys ei ole lähtenyt luomaan omaa koodigeneraattoriaan web-sovelluksille. Esimerkiksi tässä työssä käytössä ollut Anima onkin pieni startup-yritys**.** Figmaan on myös saatavilla useita eri koodigenerointilisäosia, mutta niistä suurin osa oli yhden henkilön

tekemiä. Yksi kehitysmahdollisuus opinnäytetyössä tehdylle työlle olisikin näiden erilaisten koodigenerointilisäosien kokeilu ja niiden hyötyjen selvittäminen.

Figmasta Storybookiin synkronoinnin vaikeudet on tunnistettu ongelma, johon todennäköisesti tulevaisuudessa kehitetään uudenlaisia ratkaisuja. Animalla on myös mahdollista tulevaisuudessa synkronoida Storybookista Figmaan, mutta kyseinen ominaisuus ei ollut vielä saatavilla tämän opinnäytetyön aikana. Tämä olisi myös tulevaisuudessa yksi mielenkiintoinen tutkimuskohde. Toteutetun projektin koodigeneroinnin todelliset hyödyt tultaisiin näkemään vasta myöhemmässä vaiheessa, kun komponenttikirjasto on suurempi ja siihen täytyisi tehdä säännöllisiä muutoksia.

Tekoälyn saralla on otettu monia kehitysaskeleita viime aikoina. Suurimpia tällaisia koodigeneroinnin osalta ovat esimerkiksi ChatGPT ja Github Copilot. ChatGPT:n kanssa on mahdollista kuvailla chatin välityksellä haluttua React-komponenttia, jolloin ChatGPT generoi halutun komponentin. Luontainen jatkumo tämänkaltaiselle koodigeneroinnille olisi mahdollisuus generoida koodia chatin sijasta esimerkiksi kuvista tai Figmasta. Tulevaisuudessa koodigenerointi tulee helpottamaan yhä useamman koodarin työtä, mutta samalla tulee viemään joitain junior-tason työpaikkoja, mikä nostaa rimaa työmarkkinoilla. Samalla kuitenkin syntyy myös uudenlaisia työtehtäviä, sillä jonkun täytyy kirjoittaa koneoppimisalgoritmit ja sitä varten täytyy löytyä oppimismateriaalia.

7 Yhteenveto

Tämä opinnäytetyö tutki Animan tarjoaman koodigeneroinnin hyötyä suunnittelujärjestelmän kehitystyössä ja ylläpidossa. Työssä verrattiin manuaalista koodaamista ja koodigeneroinnin hyötyjä ja haittoja. Tuloksena havaittiin, että Animan koodigenerointi vaatii suunnittelijalta ja kehittäjältä vaivannäköä ja oikeiden komponenttien määrittelyä Figmassa, jotta generoitu koodi olisi laadukasta. Kehittäjän täytyy manuaalisesti lisätä kaikki toiminnallisuudet generoituihin React-komponentteihin, mikä rajoittaa koodigeneroinnista saadut hyödyt pelkästään

ulkonäköasioihin. Työn johtopäätöksenä voidaan todeta, että Animan koodigeneroinnin käyttöönotosta tulisi tehdä projektikohtainen arvio hyötyjen ja haittojen varalta ennen käyttöönottoa.

Lähteet

Accomazzo, Anthony; Murray, Nate; Lerner, Ari. 2017. Fullstack React: The Complete Guide to ReactJS and Friends.

Adobe, 2022. Verkkoaineisto. https://xd.adobe.com/ideas/principles/design-systems/. Luettu 8.10.2022.

Anima, 2022. Verkkoaineisto. https://www.animaapp.com/. Luettu 3.10.2022.

Barr, Avron; Feigenbaum, Edward A. 1982. The Handbook of Artificial Intelligence. 295-379.

Chinda, Glad. 2022. Verkkoaineisto. https://blog.logrocket.com/what-are-react-pure-functional-components/ Luettu 08.12.2022.

Danilchenko, Yuri; Fox, Richard. 2012. Automated Code Generation Using Case-Based Reasoning, Routine Design and Template-Based Programming.

Hemmendinger, David. 2022. Verkkoaineisto. https://www.britannica.com/technology/machine-language. Luettu 19.10.2022.

Bellis, Mary. 2020. Verkkoaineisto. https://www.thoughtco.com/history-of-fortran-1991415. Luettu 14.10.2022.

Evans Data Corporation. Verkkoaineisto. https://evansdata.com/reports/viewRelease.php?reportID=9. Luettu 14.10.2022.

Figma, 2022. Verkkoaineisto. https://www.figma.com/. Luettu 3.10.2022.

Github Copilot, 2022. Verkkoaineisto. https://github.com/features/copilot. Luettu 3.10.2022.

Hui, Wendy; Chun, Kyong. 2005. On Software, or the Persistence of Visual Knowledge. Grey Room: 26-51.

IBM, 2022. Verkkoaineisto. https://www.ibm.com/cloud/blog/low-code-vs-no-code. Luettu 20.10.2022.

Izotov, Dmytro. 2020. Design System Development. Opinnäytetyö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

Java, 2022. Verkkoaineisto. https://www.java.com/en/. Luettu 3.10.2022.

Järvinen, Viljami. 2021. Automaattinen ohjelmointi käyttäen geneettistä ohjelmointia. Kandidaatintutkielma. Jyväskylän yliopisto.

NPM, 2022. Verkkoaineisto. https://www.npmjs.com/. Luettu 3.10.2022.

O'Neill Michael; Spector, Lee. 2020. Automatic programming: The open issue? Genetic Programming and Evolvable Machines, 21, 251-262.

Ojala, Tiina. 2018. Tehokkuutta ja yhtenäisyyttä suunnittelujärjestelmän avulla. Opinnäytetyö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

React, 2022. Verkkoaineisto. https://reactjs.org/. Luettu 3.10.2022.

Sass, 2022. Verkkoaineisto. https://sass-lang.com/guide. Luettu 09.12.2022.

Techcrunch, 2021. Verkkoaineisto. https://techcrunch.com/2021/09/01/anima-a-no-code-tool-that-turns-designs-into-code-raises-10-million-series-a. Luettu 20.10.2022.

Toptal, 2022. Verkkoaineisto. https://www.toptal.com/designers/ui/figma-design-tool. Luettu 8.10.2022.

Uxdesign, 2022. Verkkoaineisto. https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969. Luettu 8.10.2022.

Uxtools, 2018. Verkkoaineisto. https://uxtools.co/survey-2018/#design-system. Luettu 8.10.2022.

Uxtools, 2021. Verkkoaineisto. https://uxtools.co/survey-2021/. Luettu 8.10.2022.

Varkki, Kaarle. Verkkoaineisto. https://www.sitepoint.com/popular-react-ui-component-libraries/. Luettu 09.12.2022.

Yli-Hukkala, Aatu. 2021. React-sovelluksen teko nykyaikaisilla menetelmillä. Opinnäytetyö. Tampereen ammattikorkeakoulu. Theseus-tietokanta.