Aleksi Kosonen

Design systemin tekninen jakelu

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

11.4.2023

Tiivistelmä

Tekijä:	Aleksi Kosonen
Otsikko:	Design systemin tekninen jakelu
Sivumäärä:	40 sivua
Aika:	11.4.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaaja:	Lehtori Ulla Sederlöf

Insinöörityössä tutkittiin design systemin teknistä jakelua suunnittelijan ja ohjelmistokehittäjän näkökulmasta. Design systemit eli suunnittelujärjestelmät ovat kokoelma uudelleenkäytettäviä resursseja ja parhaita työskentelytapoja, jotka edistävät digitaalisen palvelun tai tuotteen suunnittelua ja kehitystä. Työssä keskityttiin löytämään parhaat käytännöt ja ohjelmistot uudelleenkäytettävän ja ylläpidettävän jakelukokonaisuuden rakentamiseksi.

Työn design systemiä varten tehtiin Figma-suunnittelukirjasto, josta erinäisin työkaluin johdettiin määritellyt tyyliresurssit tyylimuuttujina ja uudelleenkäytettävinä komponentteina web-kehittäjän käyttöön. Suunnittelijan tekemät muutokset suunnittelukirjastossa on jakelukokonaisuuden ansiosta vaivatonta jakaa kontrolloidusti uudelleenkäytettävien komponenttien sekä lopullisen ohjelmiston käyttöön.

Teknisen jakelukokonaisuuden rakentaminen osoitti kokonaisuuden olevan monivaiheinen ja vaativan myös eri tiimien yhteistyötä sekä ymmärrystä toistensa tarpeista. Jakelukokonaisuus on kuitenkin erittäin tarpeellinen tilanteessa, jossa useat kehitystiimit tarvitsevat samoja resursseja käyttöönsä.

Avainsanat: Design system, suunnittelujärjestelmä, käyttöliittymäsuunnittelu, ohjelmistokehitys, design token, React, TypeScript, Figma

Abstract

Author:	Aleksi Kosonen
Title:	The technical distribution of design systems
Number of Pages:	40 pages
Date:	11 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and communications technology
Professional Major:	Mobile Solutions
Supervisor:	Ulla Sederlöf, Senior Lecturer

The purpose of this bachelor's thesis was to study the technical distribution of a design system from the perspective of the designer and the software developer. Design systems are a collection of reusable components and best practices that help to design and develop digital products and services. The work aimed to find the best practices and software for building a reusable and maintainable distribution system.

A Figma design library was made for the work's design system, from which the style resources defined as style variables and reusable components for the web developer were derived using various tools. Thanks to the distribution chain, the changes made by the designer in the design library are easily distributed to the use of reusable components and the final software in a controlled manner.

In order to study the distribution system, a Figma design library was built for the design system. Using various tools, the defined Figma style resources were distributed as variables and reusable components for the web developers to use. Using the distribution system, the changes made by the designer in the design library are easily and controllably distributed to the reusable components and the final software.

The technical distribution system proved to be multi-phased and requires collaboration and understanding among different teams. However, the distribution system is necessary and valuable in a situation where several development teams need the same resources for their use.

Keywords: Design system, user interface design, software development, design token, React, TypeScript, Figma

Sisällys

Lyhenteet

CSS: *Cascading Style Sheets*. Tyylittelykieli, jolla määritellään tyylittelysääntöjä, joita dokumentti ja sen elementit tulkitsevat.

JSON: *JavaScript Object Notation*. Tiedonvälitykseen ja tallennukseen käytetty avoimen standardin tiedostomuoto.

RGB: *Red Green Blue*. Värimalli, jossa muodostetaan värejä sekoittamalla punaista, vihreää ja sinistä.

NPM: *Node Package Manager*. Maailman suurin avoimen lähdekoodin ohjelmistorekisteri, jossa eri tahot pystyvät käyttämään ja jakamaan erinäisiä koodipohjaisia paketteja.

API: *Application programming interface*. Rajapinta, jolla eri ohjelmat voivat pyytää ja vaihtaa tietoja keskenään.

1 Johdanto

Insinöörityössä perehdytään design systemien tekniseen jakeluun sekä suunnittelijan että ohjelmistokehittäjän näkökulmasta. Design systemillä tarkoitetaan hallittua kokoelmaa, joka sisältää esimerkiksi uudelleenkäytettäviä resursseja ja parhaita työskentelytapoja, jotka edistävät digitaalisen palvelun tai tuotteen suunnittelua ja kehitystä. Tarkoituksena on selvittää, miten suunnitellut resurssit saadaan hyödynnettyä ohjelmistokehittäjien työssä ja miten näitä resursseja hyödyntämällä ohjelmistokehittäjät pystyvät luomaan uudelleenkäytettäviä kokonaisuuksia. Tavoitteena on löytää sopivat ohjelmat, tekniikat ja käytännöt mahdollisimman skaalautuvaan ja uudelleenkäytettävään tekniseen jakelukokonaisuuteen ja luoda selkeitä käytäntöjä, kommunikaatiota ja teknistä yhteistyötä tiimien välille.

Design systemit pitävät sisällään käyttöliittymän suunnittelumalleja, uudelleenkäytettäviä käyttöliittymäkomponentteja, brändin viestintämalleja ja muita parhaita työskentelytapoja, jotka edistävät sovelluskehitystä eri tiimien välillä. Monesti yrityksissä, joissa kehitellään design system, työstetään useita eri sovelluksia lukuisien tiimien voimin. Näissä projekteissa todennäköisesti tavoitellaan yhtenevää ja brändinmukaista käyttäjäkokemusta, ja tästä syystä monet käyttöliittymäkomponentit ja suunnitteluratkaisut ovat samankaltaisia. Jottei jokaisen tiimin tarvitse rakentaa itse jokaista komponenttia alusta alkaen, on suositeltavaa, että useampaa tiimiä koskettavat resurssit hallitaan ja jaetaan kontrolloidusti design systemistä. Näin vältetään mahdollinen kaksinkertaisen työn riski ja rakennetaan yhdenmukaista suunnittelija-, ohjelmistokehittäjä- ja käyttäjäkokemusta.

Design systemit rakentuvat usein yhteistyöllä eri tiimien voimin, jolloin design systemin kehittämiseen ja ylläpitoon osallistuvat sitä hyödyntävät tahot. Näin rakentuu kokonaisuus, joka vastaa tiimien tarpeisiin. Tämän takia on tärkeää perehtyä siihen, mitä työvaiheita suunnittelijan ja ohjelmistokehittäjän on käytävä prosessissa läpi ja mitä näissä työvaiheissa tulee ottaa huomioon, jotta

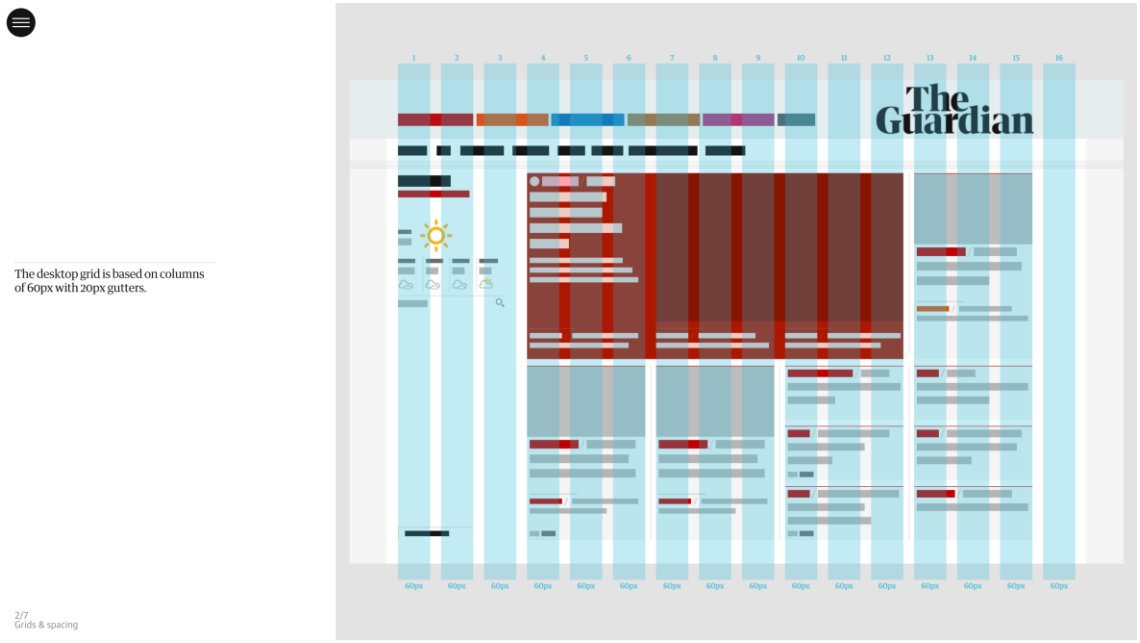
lopputuloksena on mahdollisimman toimivia ja yleiskäyttöisiä kokonaisuuksia, joita design systemistä voidaan jakaa.

2 Design systemit

2.1 Design systemit – brändin, kehityksen ja käyttäjäkokemuksen fuusio

Design systemit ovat kokonaisuuksia, jotka edistävät sovellusten yhdenmukaista ja nopeaa suunnittelua ja kehitystä. Design systemit on usein mielletty pelkiksi suunnittelijoiden käyttämiksi komponenttikirjastoiksi, joita hyödyntämällä voidaan suunnitella sovellusten käyttöliittymiä. Nykyään yhä useammin design systemit nähdään laajempana työkaluna, joka auttaa sovelluskehitystä pelkkää komponenttikirjastoa laajemmin.

Design systemien on nykyään kuvailtu rakentuvan konkreettisista ja abstrakteista osista. Konkreettisina osina design systemiä voi mieltää esimerkiksi suunnittelijoille ja ohjelmistokehittäjille tarkoitetut materiaalit kuten typografiat, valmiit käyttöliittymäkomponentit sekä erinäiset suunnittelumallit (kuva 1). Abstraktit osat kuvastavat syvemmin esimerkiksi organisaation brändin viestintämalleja, parhaita työskentelytapoja ja ajatusmalleja. Design system voidaankin nähdä organisaation digitaalisen kehityksen keskitettynä tiedonlähteenä. (1.)



Kuva 1. The Guardianin design systemin suunnittelumalli sivuston eri osioiden ja elementtien asetteluun (2).

Design systemit nopeuttavat työtä ja vähentävät kaksinkertaisen työn riskiä, sillä jokaisen sitä hyödyntävän tiimin ei tarvitse aloittaa suunnittelu- tai kehitystyötään tyhjästä. Design system tarjoaa hyvän lähtötilanteen tiimien työhön tarjoamalla esimerkiksi brändinmukaiset suunnittelumallit ja ohjelmistokehityksessä hyödynnettäviä rakennuspalasia. Tätä väitettä tukee IBM:n Carbon design systemiin perustuva kyselytutkimus (taulukko 1), jonka mukaan ohjelmistokehittäjän työ oli 47 % nopeampaa design systemin avulla kuin ilman sitä (3). Design systemit helpottavat myös muun muassa uusien työntekijöiden ja muiden kehitykseen osallistuvien tahojen perehdytystä ja koulutusta sekä selkeyttävät työtä määrittelemällä yhteisiä kehityskohteita. Kyselytutkimuksen mukaan design systemin kokee erittäin mieluisaksi tai mieluisaksi työvälineeksi 65 % vastaajista (4).

Taulukko 1. Sparkboxin teettämän kyselyn vastausmäärät kysymykseen, mistä resursseista design systemit vastanneiden organisaatioissa koostuvat (3).

Resurssi	**Määrä**

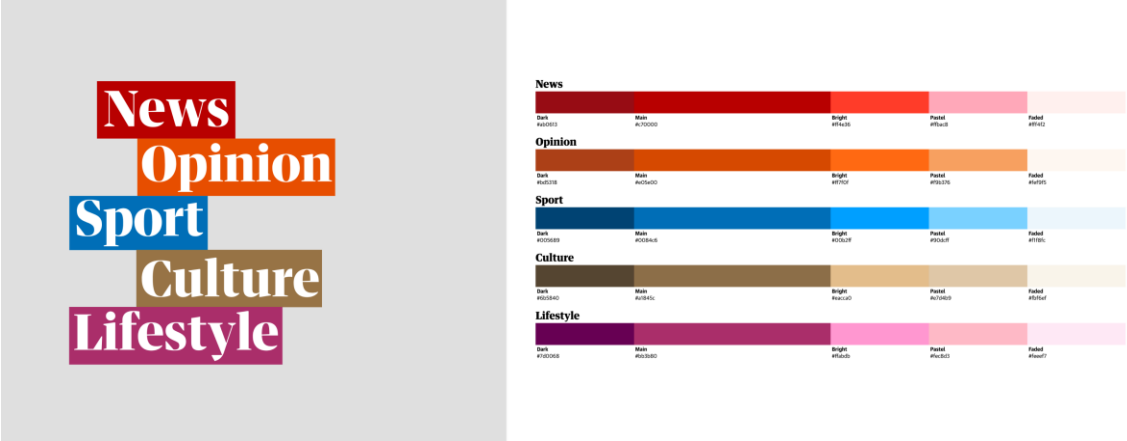
Typografia, värit ja tyylit	98 %
Komponentit	95 %
Komponenttidokumentaatio	85 %
Valmiit resurssit suunnittelijalle	75 %
Ruudukkoresurssit	71 %
Design tokens	69 %
Koodiresurssit ohjelmistokehittäjälle	65 %
Saavutettavuusohjeistukset	57 %

Yrityksissä ymmärretään nykyään yhä paremmin, ettei hieno ja hyvin toimiva käyttöliittymä ole pelkästään visuaalisesti miellyttävä tekijä, vaan elintärkeä osa nykypäivän liiketoimintaa. Kinesis Inc:n mukaan 94 % käyttäjien ensivaikutelmista sivustoilla liittyy käyttöliittymän suunnitteluun ja 75 % käyttäjistä teki päätelmiä yrityksestä käyttäjäkokemuksen mukavuuden perusteella. Hyvällä ja johdonmukaisella käyttäjäkokemuksen suunnittelulla ja toteutuksella on siis selkeää markkinaetua kilpailijoihin nähden, ja laadukkaaseen ja tehokkaaseen sovelluskehitykseen on syytä panostaa. (5.)

Design system auttaa yrityksiä rakentamaan yhdenmukaista käyttäjäkokemusta tarjoamalla samat resurssit koko organisaation käyttöön. Näin varmistetaan,

että jokaisen eri tuotteen loppukäyttäjät kokevat yhdenmukaisuutta, tunnistettavuutta ja saavutettavuutta muiden organisaation tuotteiden ja palveluiden kesken (6). Design system on siis viitekehys, joka vahvistaa loppukäyttäjän kokemusta organisaation brändistä.

Esimerkki design systemin tuomasta hyödystä ja selkeydestä suunnitteluun ja käyttäjäkokemukseen on The Guardianin design systemin esittelemä väripaletti (kuva 2). Siinä on jokaiselle The Guardianin digitaalisen uutispalvelun nimetylle uutiskategorialle omistettu värinsä ja näiden värien variantit tumma, pääväri, kirkas, pastelli ja häivytetty. Värejä, jotka nimetään niiden käyttötarkoituksen mukaan, voidaan kutsua semanttisiksi väreiksi (7). Yksinkertaisella semanttisella värijaolla suunnittelija tietää, mitä väriä hänen tulee käyttää suunnitellessaan eri käyttöliittymiä uutiskategorioille.



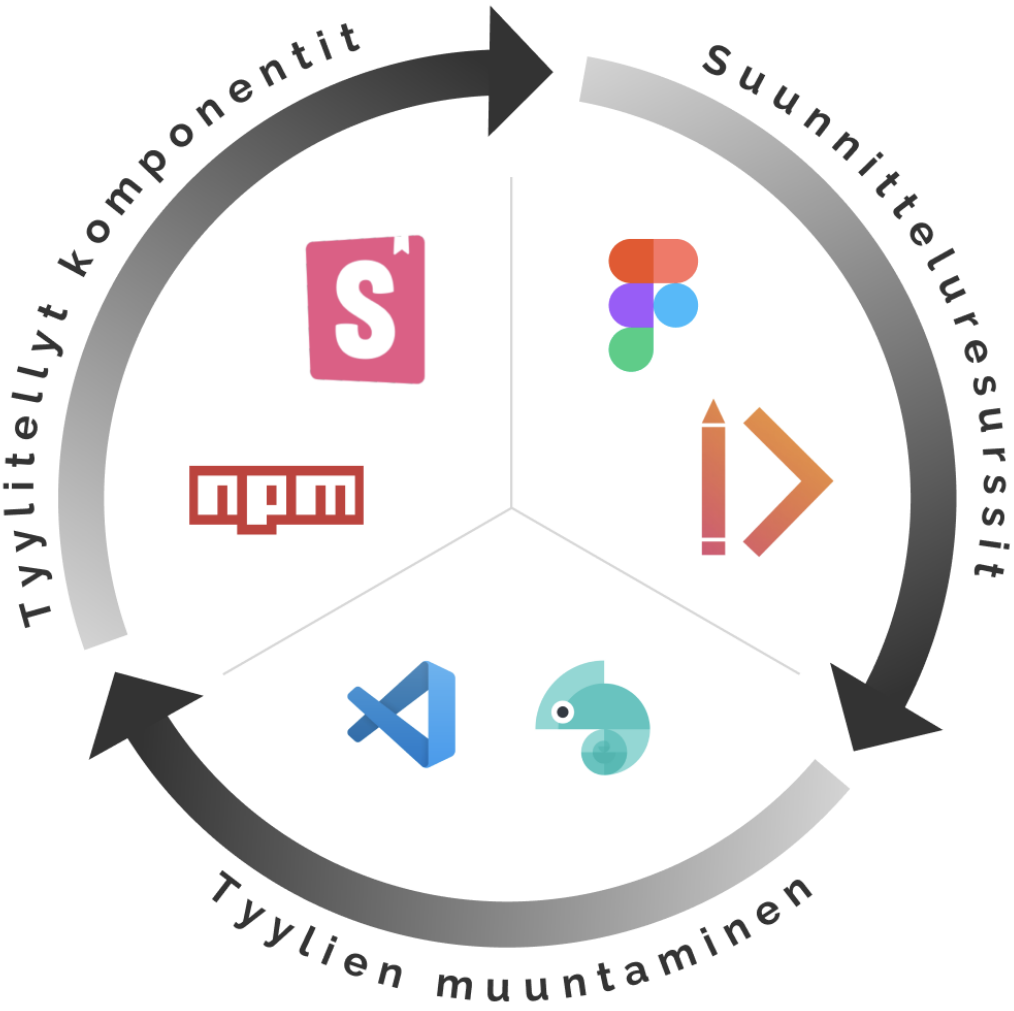
Kuva 2. The Guardianin design systemin väripaletti, jossa jokaiselle uutiskategorialle annetaan nimetty värinsä (2).

Värien selkeä erottelu käyttötarkoituksen mukaan luo myös käyttäjälle mahdollisuuden oppia erottamaan eri kategoriat visuaalisesti toisistaan. Värien nimeäminen käyttötarkoituksen mukaan mahdollistaa myös värien selkeämmän viestimisen tiimien välillä. Sen sijaan, että suunnittelija puhuisi pelkästään esimerkiksi sinisestä väristä, voi hän puhua väristä nimeltä urheilu. Näin design system parhaimmillaan rakentaa koherenttia suunnittelija- ja käyttäjäkokemusta yksinkertaisella semanttisella väripaletilla.

2.2 Insinöörityön design system ja tavoitteet sen jakeluun

Insinöörityössä haluttiin perehtyä tarkemmin design systemin teknisiin osa-alueisiin. Tavoitteena oli löytää parhaat ohjelmistot, teknologiat ja työskentelytavat mahdollisimman uudelleenkäytettävän ja skaalautuvan teknisen kokonaisuuden saavuttamiseksi.

Insinöörityön design system suunniteltiin pitämään sisällään The Guardianin design systemin kaltaisen, mutta kuitenkin typistetymmän väripaletin, jossa olisi fiktiivisen organisaation brändin mukainen värimaailma. Näitä värejä hyödyntämällä suunniteltaisiin yksinkertainen käyttöliittymäkomponentti, joka toimisi pohjana tekniselle toteutukselle. Komponentti oli tarkoitus ohjelmoida ja julkaista osana komponenttikirjastoa, jota eri kehitystiimit pystyisivät hyödyntämään. Insinöörityö oli tarkoitus perehtyä parhaisiin työskentelytapoihin niin komponentin suunnittelussa, dokumentoinnissa, kehityksessä kuin komponenttikirjaston julkaisemisessa. Kuvassa 3 havainnollisestaan työn eri vaiheet sekä niihin tarvittavat teknologiat.



Kuva 3. Design systemin työvaiheet ja työvaiheisiin tarvittavat teknologiat.

Insinöörityön design systemin kattamat työnkuvat tulisivat olemaan suunnittelija ja ohjelmistokehittäjä. Suunnittelija vastaisi suunnitteluresurssien kuten väripaletin ja käyttöliittymäkomponentin suunnittelusta. Ohjelmistokehittäjä puolestaan vastaa suunnitteluresurssien teknisestä toteutuksesta mahdollisimman uudelleenkäytettävässä muodossa, josta muut organisaation ohjelmistotiimit voivat hyötyä.

Suunnitteluresurssien toteutus oikeiksi komponenteiksi ei aina tapahdu ongelmitta, ja joskus komponentteja ei ole toteutettu ohjelmistoissa suunnitellun mukaisesti (8). Tämän takia erittäin tärkeä liitoskohta design systemin kannalta

on suunnittelijan ja ohjelmistokehittäjän roolien välillä. Insinöörityössä perehdytään siihen, miten suunnittelijan luomat tyylit saadaan muunnettua ohjelmistokehittäjälle helposti käytettävään muotoon virheiden välttämiseksi ja miten mahdolliset suunnittelijan tekemät muutokset jakautuisivat automaattisesti ohjelmistokehittäjien tai suoraan olemassa olevien komponenttien käyttöön.

Dokumentoimalla eri työnkuvien työvaiheet design systemiin organisaatio tarjoaa valmiit työskentelymallit työntekijöille. Näistä työskentelymalleista, tekniikoista ja käytännöistä rakentuisi lopulta itseään ylläpitävä tekninen jakelukokonaisuus, jonka lopputuotteena olisi käyttöliittymäkomponentteja suoraan sovelluskehityksen parissa työskenteleville ohjelmistokehitystiimeille. Eri työroolien vaikuttavuuden dokumentointi jakelukokonaisuuden eri osiin edistäisi yhteisöllisyyttä ja helpottaisi kommunikaatiota eri tiimien välillä. Myös mahdolliset muutokset jakautuisivat suoraan jokaiseen tiimiin työvaiheiden avulla.

2.3 Design systemin tyylien luominen Figmassa

Kun design systemin suunnitteluresursseja lähdetään työstämään, on hyvä valita suunnitteluohjelmisto, joka tukee helppokäyttöisyyttä, uudelleenkäytettävyyttä, jatkokehitystä ja keskitettyä kokonaisuutta. Markkinoilla on useita erilaisia työkaluja suunnitteluresurssien rakentamiseen, kuten Figma, Sketch ja Adobe XD.

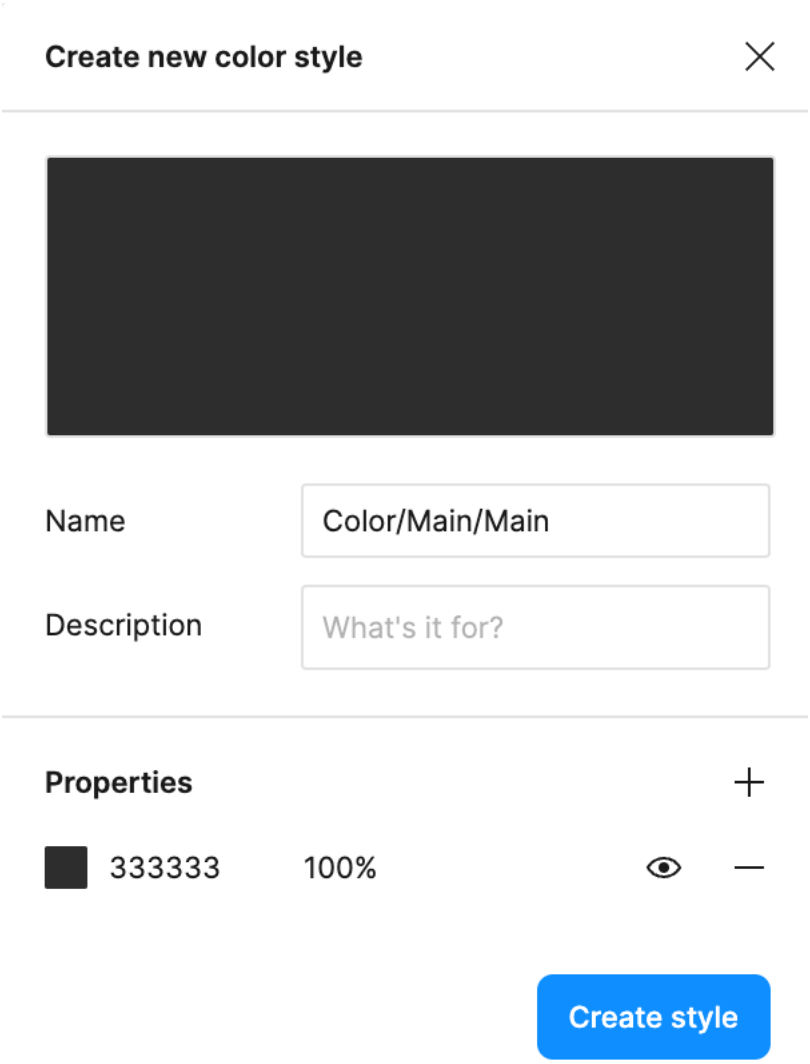
Tässä insinöörityössä toteutettiin design systemin suunnittelukokonaisuus suunnittelutyökalu Figmalla. Se on pääosin suunnittelijoille suunnattu työkalu, joka on täynnä toiminnallisuuksia, jotka tukevat suunnittelun yhteistyötä työtiedostoissa, kuten suunnittelutiedostojen yhtäaikainen muokkaaminen, kommentointi ja versionhallinta. Figmalla pystyy luomaan niin design systemiin tarvittavia määriteltyjä tyylejä, uudelleenkäytettäviä suunnittelukomponentteja kuin kompleksisia lopullisia prototyyppisuunnitelmia. (9.)

Suunnittelijan näkökulmasta design systemit koostuvat teknisesti tyyleistä, kuten väreistä, typografioista ja efekteistä, joista komponenttien ja isompien

suunnittelukokonaisuuksien voidaan nähdä rakentuvan (10). Tyylien luominen on ensimmäinen askel kohti uudelleenkäytettävää ja skaalautuvaa kehitystä, sillä määriteltyjen tyylien luominen mahdollistaa niiden dokumentaation, jakelun ja ylläpidon (11). Keskitetysti jaetut ja hyvin dokumentoidut tyylit helpottavat suunnittelijan työtä tarjoamalla esimerkiksi käyttövalmiit brändinmukaiset värit sekä käyttötarkoitukset eri väriarvoille.

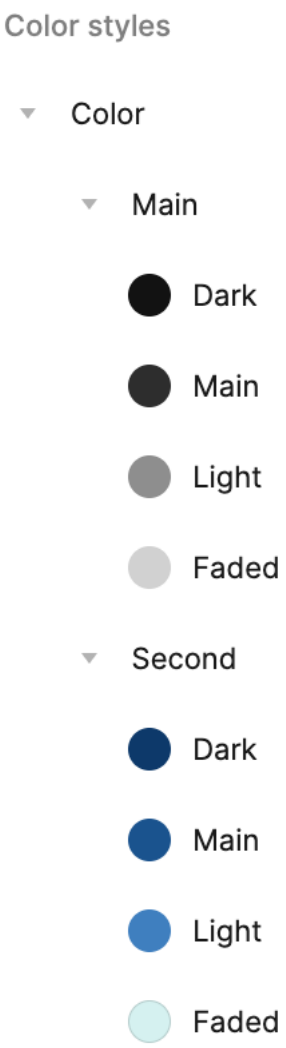
Insinöörityössä perehdytään siihen, miten käyttöliittymäkomponentteja suunnitellaan käyttämällä Figmaan luotuja värityylejä, efektejä ja typografioita ja miten ne saadaan vietyä ohjelmistokehitykseen mahdollisimman skaalautuvasti. Jokaisen komponentin on hyvä rakentua määritellyistä tyyleistä, jotta ylläpito ja muutostilanteet, kuten esimerkiksi komponentin taustavärin muutos, olisivat mahdollisimman helppoja Sen sijaan, että jokaisen muuttuneen taustavärin sisältävän komponentin väriarvoa tarvitsisi manuaalisesti muokata, voidaan muokata vain tyylin arvoa, jolloin muutokset jakautuvat jokaiseen komponenttiin, jossa tyyliä on hyödynnetty. Jotta käyttöliittymäkomponentteja saadaan suunniteltua tyyleillä, perehdytään ensin siihen, miten määritellyt tyylit rakentuvat Figmassa. Insinöörityön design system Figma-kirjastoa varten tehtiin uusi Figma Team, jonne luotiin uusi suunnittelutiedosto. Figma-tiimiin pystyy kutsumaan muita suunnittelijoita hallitusti, jotta tarvittavat tiimit pääsevät käsiksi design systemin resursseihin (12).

Design systemin värityylit luotiin hyödyntäen Figman Color Styles -toimintoa, jolla suunnittelija pystyy luomaan projektissa käytetystä väristä määritellyn värityylin (kuva 4).



Kuva 4. Värityyli-toiminto Figmassa ja halutun rakenteen saavuttava syntaksi.

Tavoitteena oli luoda samantyyppinen väripaletti kuin aikaisemmassa The Guardianin design system -esimerkissä. Jotta värit, värien semanttiset tasot ja tasojen variantit saadaan eroteltua toisistaan, hyödynnettiin Figman Color styles -toiminnon kategorisoinnin toiminnallisuutta. Jokainen sana vinoviivalla eroteltuna jakautuu omaksi kategoriakseen niin Figman sisällä kuin myös myöhemmin jakeluvaiheessa (13). Tasojen nimeämiseen käytettiin Main- ja Second-nimityksiä, ja tasoille tehtiin variantit dark, main, light ja faded. Näin esimerkiksi brändin päävärin luominen vaati Color style työkaluun esimerkkikuvan 5 mukaisen syötteen ”Color/Main/Main”.



Kuva 5. Lopulliset värityylit Figmassa.

2.4 Design systemin dokumentointi Zeroheightiin

Jotta design system olisi muutakin kuin komponenttikirjasto suunnittelijalle, on design systemille suositeltavaa rakentaa erillinen dokumentaatio. Dokumentaation on hyvä kattaa suunnittelijoiden ja kehitystiimien tarpeiden lisäksi myös muiden design systemin parissa työskentelevien tiimien tarpeet. Näin dokumentaatiolla pystytään yhdistämään eri tiimejä, jotka työskentelevät design systemin parissa sekä ennaltaehkäisemään eri tiimien tiedonlähteiden pirstaloitumista.

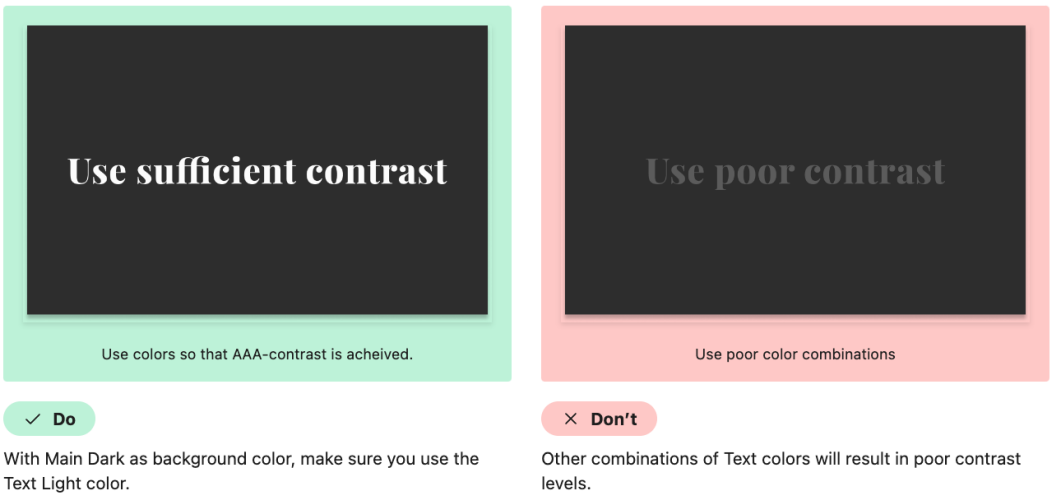
Yleinen työkalu dokumentaatioiden rakentamiseen on dokumentointipalvelu Zeroheight (14), jota käyttävät design systemin dokumentoinnissaan muun muassa Adobe, The Guardian ja Red Bull. Zeroheight on keskiö suunnittelijoille ja ohjelmistokehittäjille sekä tuote- ja markkinointitiimeille. Se yhdistää jokaisen, joka työskentelee design systemin parissa. Zeroheightin dokumentaation pystyy jakamaan joko suljetusti tuotetiimien kesken tai julkisesti. (14.)

Dokumentaatio voidaan nähdä seuraavana konkreettisena askeleena kohti suunnittelun ja kehityksen yhteyden vahvistamista värityylien luomisen jälkeen. Suunnittelijan näkökulmasta dokumentaatiossa on hyvä pyrkiä vastaamaan kysymyksiin miten, miksi ja milloin tiettyä suunnitteluresurssia tulee käyttää, ja ohjelmistokehittäjälle dokumentaatio antaa resurssit, joilla toteuttaa suunnitellut kokonaisuudet (15). Koska suunnitteluun ja ohjelmistokehitykseen liittyvät ohjeistukset, käytännöt ja resurssit on hyvä saada kontrolloidusti samaan dokumentaatioon, on hyvä valita työkalu, joka integroituu helposti niin Figmaan kuin mahdollisiin ohjelmointityökaluihin kuten Storybookiin. Tästä syystä Zeroheight (14) valittiin käyttöön tässä insinöörityössä.

Koska Zeroheight on suunniteltu integroitumaan Figman kanssa, oli design systemin suunnitteluresurssien tuominen Zeroheightiin vaivatonta. Lisäämällä Zeroheightin latausresursseihin insinöörityön design systemin Figma-suunnittelutiedoston linkki saatiin luotua uusi Zeroheight-dokumentointikokonaisuus vaivattomasti suoraan palvelun etusivulta (kuva 6). Tämän jälkeen Zeroheight on yhdistynyt design systemin Figma-tiedostoon ja sen dokumentoiminen voitiin aloittaa. Ajantasaiset resurssit Figmasta saadaan päivittämällä kyseinen latausresurssi Zeroheightin valikosta.

Kuva 6. Design systemin Figma-tiedoston linkittäminen Zeroheightiin.

Zeroheightiin voi dokumentoida tarkemmin esimerkiksi, miten komponentit rakentuvat, mitä ominaisuuksia niillä on, miten niitä tulisi käyttää suunnittelussa ja miten niiden lopullinen koodiratkaisu toimii. Visuaaliseksi dokumentaatioesimerkiksi rakennettiin aikaisemmin luodun päävärityylin käyttöesimerkki. Esimerkissä havainnollistettiin, mitä tekstiväriä tulee käyttää saavuttaakseen riittävän kontrastitason, jos design systemin pääväri on asetettu taustaväriksi (kuva 7). Tarjoamalla visuaalisia ja selkeitä käytännön esimerkkejä helpotetaan suunnittelijan työtä ja luodaan visuaalisesti brändin ilmettä myös dokumentaatioon.



Kuva 7. Dokumentointiesimerkki Zeroheightissä aikaisemmin luodulla primäärivärillä.

3 Design tokenit kehityksen ja suunnittelun linkkinä

3.1 Design tokenit ja niiden rakenne

Yksi tärkeistä liitoskohdista sovelluskehityksessä on suunnittelutiimin ja ohjelmointitiimin yhteistyö. Design systemien työskentelytapojen hallinnan näkökulmasta tämä on otollinen liitoskohta, jotta tiimien välille saadaan rakennettua hyviä käytäntöjä ja työmalleja. Tästä syystä suunnittelijoiden ja ohjelmistokehittäjien tulee miettiä yhdessä, miten suunnitellut kokonaisuudet saadaan implementoitua ohjelmistokehityksessä mahdollisimman pienellä vaivalla. Oleellinen tekijä tässä vaiheessa ovat design tokenit. Ne ovat väreistä, typografioista ja efekteistä johdettuja tyyliarvoja, jotka voidaan muuttaa eri ohjelmistoteknologioiden käyttöön. (16.)

Design tokenit kuvastavat kaikkia niitä arvoja, joita tarvitaan design systemin rakentamisessa ja ylläpitämisessä (17). Design tokenit rakentuvat design systemin suunnittelutiedostossa määritellystä tyyleistä, joista ne pystytään erinäisten ohjelmistojen avulla kääntämään ohjelmistokehittäjille hyödylliseen muotoon, kuten JSON tai CSS. Design tokenien oikeaoppisen käytön voidaan nähdä luovan dialogia eri tiimien välille ja tuovan johdonmukaisuutta tyyliratkaisuihin.

Aluin perin design tokenit on kehittänyt Salesforce-nimisen yrityksen design systemin kehityksestä vastaava tiimi. Erityisesti Jina Anne on mainittu design tokenien kehittäjäksi. Anne kuvailee itse design tokeneita ja niiden käyttöä metodologiaksi sen sijaan, että tokenit olisivat pelkkiä muuttujia. Jos design tokenit näkisi vain muuttujina, se olisi sama kuin mieltäisi responsiivisen suunnittelun pelkiksi mediasäännöiksi. (18.)

Rakenteellisesti design tokenit koostuvat avain- ja arvopareista (esimerkkikoodi 1). Nämä avain- ja arvoparit voidaan tallentaa esimerkiksi JSON-muodossa omaan tiedostoon. Näitä tiedostoja pystyy myöhemmin prosessoimaan

erinäisillä tavoilla ja hyödyntämään ohjelmistojen kehityksessä eri alustoilla. Esimerkiksi samat design tokenit on helppo prosessoida eri teknologioiden tarpeeseen, kuten web-kehitykseen ja esimerkiksi Android- ja iOS-kehitykseen, jotka tarvitsevat erilaisia tyyliresursseja toimiakseen. Täten alkuperäistä design tokenit sisältävää tiedostoa muokkaamalla muokkaa jokaisen sitä hyödyntäneen ohjelmiston ulkoasua, alustasta riippumatta. (19.)

```
"Color": {
    "Main": {
      "Dark": {
        "value": "#111111"
      },
      "Main": {
        "value": "#333333"
      },
      "Light": {
        "value": "#999999"
      },
      "Faded": {
        "value": "#d7d7d7"
      }
    },
```

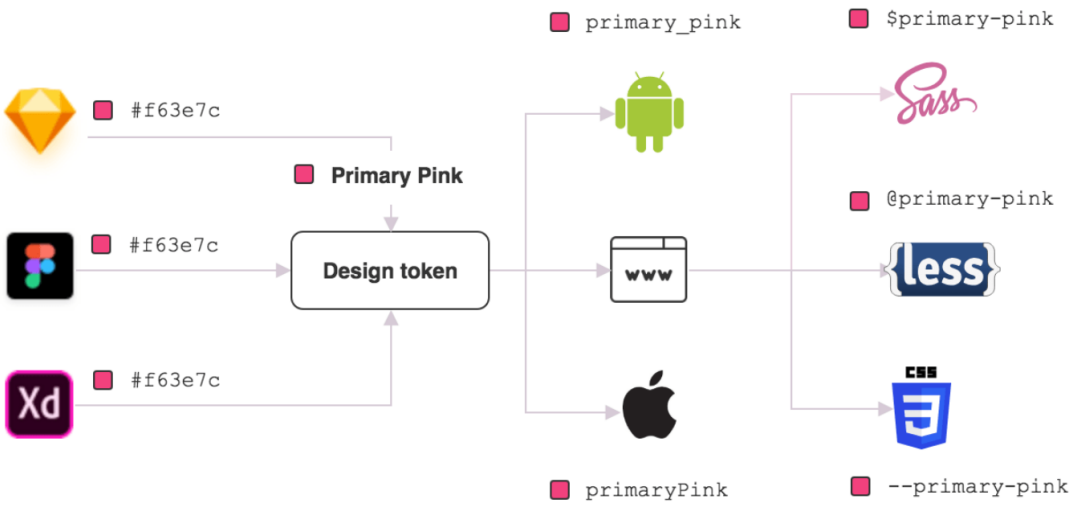
Esimerkkikoodi 1. JSON-muotoinen design token, joka pitää sisällään yrityksen päävärin eri varianttien avain- ja arvoparit.

Määrittelemällä tyylit ja generoimalla design tokenit niiden mukaan suunnittelija pystyy auttamaan kehitystiimin työtä poistamalla tarpeen käyttää kovakoodattuja tyylimääritelmiä tyylittelytiedostoissa. Esimerkiksi brändin pääväriin viitatessa ohjelmistokehittäjä voi käyttää kovakoodatun heksadesimaali- tai RGB-arvon sijaan design tokeneista johdettuja nimettyjä määreitä. Koska jokainen kehitystiimi pystyy hyödyntämään tyyliensä määrittelyyn samoja design tokeneita, ovat nämä arvot myös koherentit jokaisen kehitystiimin kesken. Tämä luo myös yhdenmukaisuutta tiimien väliseen kommunikaatioon, sillä nimitykset ovat yhtenevät jokaisen tiimin kesken. (20.)

Design tokenit auttavat myös suunnittelumuutosten toteuttamisessa. Muokkaamalla määriteltyjä tyylejä design systemin suunnittelutiedostossa suunnittelija muokkaa samalla myös siitä generoidun design tokenin arvoa. Esimerkiksi jos brändin sekundäärinen värisävy mielletään liian tummaksi ja sen

väriarvoa päätetään muuttaa, ei ohjelmistokehittäjän tarvitse kuin päivittää design tokens -tiedosto ohjelmistossa ja päivittyneet tyylit ovat suoraan näkyvissä. Tämä poistaa ohjelmistokehittäjän tarpeen käydä muokkaamassa jokaista tyyliä manuaalisesti.

Design tokeneiden tarjoaman hyödyn voi siis nähdä kaksijakoisesti. Ne tarjoavat yhtenevän nimityksen ja identiteetin suunnitteluratkaisuihin alustasta riippumatta (kuva 8). Tällä tavoin jokaisessa yrityksen tuotteessa esimerkiksi sama väri on nimetty samalla tavalla ja käsitys brändin suunnitteluratkaisuista vahvistuu. Toisena hyötynä on jatkuvuus kehityksessä: mikäli design tokeneita on hyödynnetty kaikkialla, tokenin arvon muuttuessa se muuttuu koherentisti kaikkialla. (16.)



Kuva 8. Design tokenit toimivat linkkinä suunnittelijoiden ja kooditiimien välillä (17).

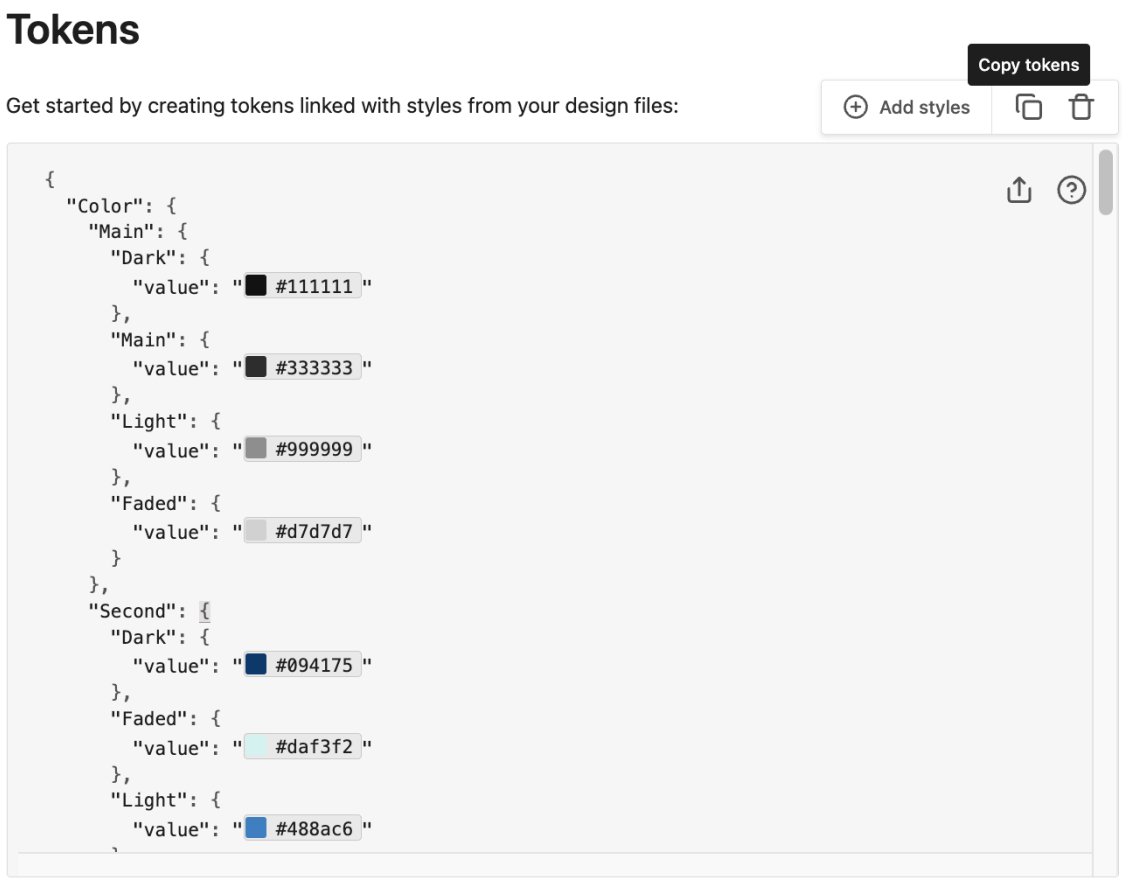
3.2 Design tokeneiden generoiminen ja jakelu

Design tokenien generoiminen ja ylläpitäminen on yleensä suunnittelutiimin vastuulla, sillä ne johdetaan suunnittelijoiden määrittelemistä tyylimääreistä. Design tokenit on järkevää rakentaa osaksi dokumentaatiota, jotta ne ovat saatavilla kontrolloidusti jokaisen tiimin kesken. Suunnittelutyökaluja design

tokenien generoimiseen on useita, kuten Sketch, Adobe XD ja Figma. Tässä insinöörityössä design tokenien generoimiseen hyödynnetään suunnittelusta tuttua suunnittelutyökalu Figmaa sekä dokumentointiin hyödynnettyä Zeroheight-dokumentointialustaa.

Kun design systemin Figma-suunnittelutiedosto on lisätty Zeroheightin latauksiin, luo Zeroheight automaattisesti Figmassa määritellyt tyylit omiksi design tokeneikseen. Design tokenit rakentuvat kategorisesti tyylien mukaan, joten tyylien generointi aikaisemman esimerkin mukaan on oleellista teknisen jakelun kannalta.

Zeroheight tarjoaa automaattisesti dokumentaatiossa oman design tokens-sivuston, johon suunnittelutiimi voi rakentaa dokumentaation design tokeneille. Design tokenit pystyy kopioimaan Zeroheightin dokumentaatiosta suoraan JSON-muotoisina (kuva 9) ja liittää omaan ohjelmistoprojektiin. Vaihtoehtoisesti Zeroheightistä voi välittää tokenit myös oman rajapinnan kautta hyödyntäen Zeroehghtin tarjoamaa url-osoitetta.



Kuva 9. Design tokeneiden kopiointi Zeroheightistä.

Eri kehitystiimit pystyisivät hyödyntämään design tokeneita suoraan Zeroheightista joko kopioimalla tiedoston manuaalisesti tai rajapinnasta url-osoitteen kautta suoraan lähdekoodissa. Näin kehitystiimit saisivat design tokenit ohjelmistonsa projektiin suoraan ja pystyisivät omilla määritelmillä muuntamaan design tokenit oman tuotteensa teknologioiden vaatimusten mukaisiksi.

Tässä insinöörityössä haluttiin kuitenkin tutkia jakelukokonaisuutta pidemmälle, jotta suunnittelijat ja ohjelmistokehittäjät pystyisivät yhdessä rakentamaan design systemin jakelukokonaisuutta eheämmäksi kokonaisuudeksi. Sen sijaan, että kehitystiimit olisivat itse vastuussa design tokenien muuntamisesta omien teknologioidensa tarpeisiin, haluttiin tässä insinöörityössä rakentaa design tokeneista käyttövalmiit tyylimuuttujat tiimien käyttöön. Tämä vähentäisi esimerkiksi uuden kehitystiimin tarvetta määritellä tyylimuuttujia, ja sen sijaan tiimillä olisi brändinmukaiset tyyliresurssit valmiina käytössä alusta asti. Näin myös muutosten toteuttaminen olisi hallitumpaa, sillä päivitetyistä design tokeneista johdettuja tyylimuuttujia pystyttäisiin testaamaan paremmin, ennen kuin päivitetty julkaisu olisi käytössä.

3.3 Design tokeneiden muuntaminen tyylimuuttujiksi

Design tokenien käyttö suoraan koodissa on mahdollista, mutta yksi niiden suurimmista teknisistä hyödyistä on niiden helppo prosessointi eri teknologioiden vaatimaan muotoon. Koska design systemin parissa voi työskennellä lukuisilla eri teknologioilla työskenteleviä ohjelmistokehitystiimejä, on suositeltavaa jakaa design systemistä valmiita tyylimuuttujia eri teknologioiden tarpeisiin.

Yksi suurimmista teknisistä hyödyistä dokumentoinnin rakentamisessa Zeroheightiin on sen tarjoama API design tokeneille. Design tokenit ovat luettavissa suoraan omasta url-osoitteesta JSON-muodossa, josta ne pystytään lukemaan suoraan omaan tiedostoon, josta niitä pystyy erinäisin kirjastoin

muuntamaan. Tämä poistaa tarpeen käydä kopioimassa design tokeneita Zeroheightin sivustolta. Tätä insinöörityön teknistä osiota varten luotiin tietokoneen hakemistoon kansio, joka sisältäisi sekä ohjelman design tokenien muuntamiselle Zeroheightin url-osoitteesta suoraan käyttövalmiiksi muuttujiksi että lopullisen komponenttikirjaston, johon perehdytään myöhemmin työssä.

Tekninen osio aloitettiin tekemällä ohjelman design tokenien muuntamiselle. Työssä alustettiin projekti tyhjässä kansiossa komennolla ’npm init -y’, joka alustaa projektiin tarvittavan package.json-tiedoston. Komennon jälkiosa ’-y’ vastaa komentorivin kyselyihin automaattisesti kyllä (21). Tämän jälkeen lisättiin tarvittavat riippuvuudet komennolla ’npm i express cors fs-extra node-fetch style-dictionary’. Nämä riippuvuudet mahdollistivat url-osoitteen lukemisen, tiedoston kirjoittamisen sekä design tokenien muuntamisen tyylimuuttujiksi.

Jotta design tokenit saadaan muunnettua ohjelmakoodille luettavaan muotoon, on suositeltavaa, että ne prosessoidaan juuri asennetulla Style Dictionary -paketilla. Style Dictionary on dokumentaation kuvauksen mukaan järjestelmä, jonka avulla voi määrittää tyylit kerran tavalla, joka sopii jokaiselle alustalle (22). Style Dictionaryn määritelmäkoodilla ohjelmistokehittäjä pystyy valitsemaan tarvittavan teknologian, jonka formaatille järjestelmä kääntää tokenit. Esimerkkinä näistä teknologioista, joille Style Dictionary voi tokenit kääntää, ovat SASS, Swift ja Flutter. Yksi web-kehityksen peruspilareista on tyylittelykieli CSS, jolla pystyy määrittelemään tyylittelysääntöjä, joita dokumentti ja sen elementit tulkitsevat (23). Koska myöhemmin kehiteltävät komponentit tulivat olemaan web-pohjaisia, muunnettiin design tokenit tässä insinöörityössä CSS-muuttujiksi.

Ohjelmaan lisättiin esimerkkikoodin 2 mukainen index.js-tiedosto.

```
import express from 'express'
import cors from 'cors'
import fetch from 'node-fetch'
import fs from 'fs-extra'
import StyleDictionary from 'style-dictionary'

const PORT = process.env.PORT || 3001

const app = express()
app.use(cors())

fetch('<your-zeroheight-token-url>')
  .then((response) => {
    return response.json()
  })
  .then((json) => {
    fs.writeFile('./ds-tokens.json', JSON.stringify(json), (err) => {
      const styleDictionary = StyleDictionary.extend({
        source: ['ds-tokens.json'],
        platforms: {
          css: {
            transformGroup: 'css',
            buildPath: '../ts-storybook-components/stories/',
            files: [
              {
                destination: 'variables.css',
                format: 'css/variables',
              },
            ],
          },
        },
      })
      styleDictionary.buildAllPlatforms()
      if (err) {
        throw new Error('Error reading tokens.')
      }
    })
  })

app.listen(PORT, () => {})
```

Esimerkkikoodi 2. index.js-tiedosto, joka tallentaa Zeroheightista design tokenit json-tiedostoksi ja generoi komponenttikirjastoon CSS-muuttujat sisältävän tyylittelytiedoston.

Käyttämällä Zeroheightin tarjoamaa url-osoitetta design tokenit haetaan ja tallennetaan omaan tiedostoon ’ds-tokens.json’. Jos tämän tiedoston luominen onnistuu, käyttää ohjelma Style Dictionary npm-pakettia ja muuntaa design tokenit määrättyyn CSS-muotoon. Tässä esimerkissä pystyttiin jo määrittelemään uuden tyylittelytiedoston sijainti. Koska komponenttikirjasto tultiin rakentamaan samaan kansioon, voitiin määritellä tyylittelytiedoston generoituvan suoraan komponenttikirjastoon määrittelemällä koodissa

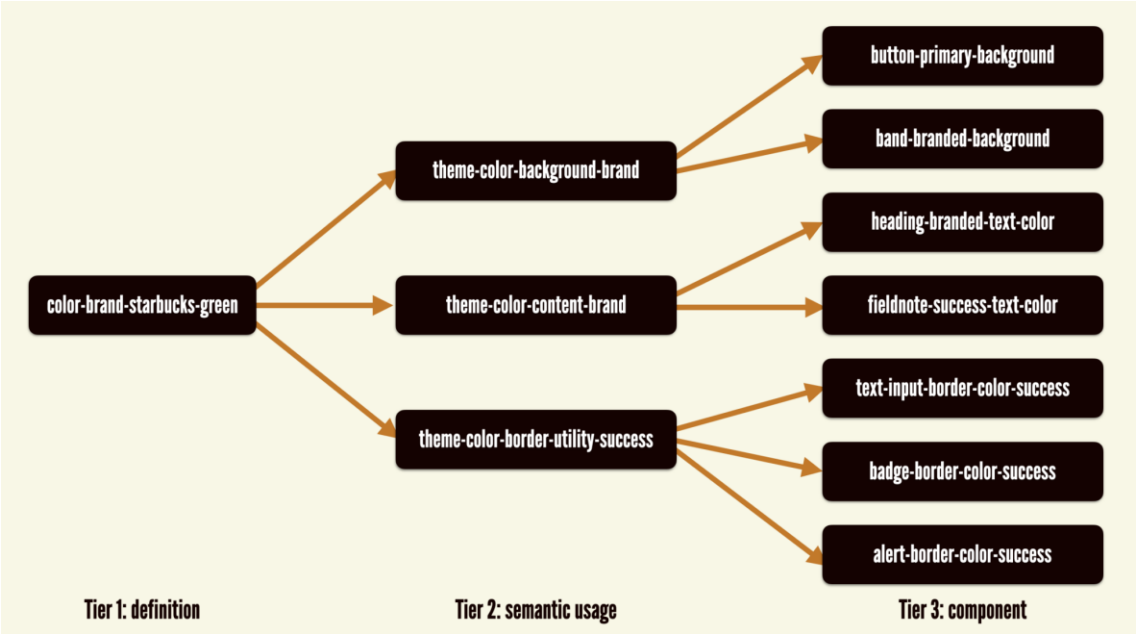
’buildPath: '../ts-storybook-components/stories/’’. Tällä tavoin suorittaessa “index.js”-tiedoston koodin saa tuleva komponenttikirjasto projektissa ’”ts-storybook-components” suoraan ajantasaiset tyylimuuttujat käyttöönsä (kuva 10).

```
:root {
  --color-main-dark: #111111;
  --color-main-faded: #d7d7d7;
  --color-main-light: #999999;
  --color-main-main: #333333;
  --color-second-dark: #094175;
  --color-second-faded: #daf3f2;
  --color-second-light: #488ac6;
  --color-second-main: #1c5d99;
  --color-text-dark: #333333;
  --color-text-faded: #d7d7d7;
  --color-text-light: #fcfcfc;
  --color-text-main: #666666;
  --typography-body-large-font-family: 'Raleway', 'RalewayRoman-Medium';
  --typography-body-large-font-size: 18px;
  --typography-body-large-letter-spacing: 0px;
  --typography-body-large-line-height: 135%;
  --typography-body-large-font-weight: 500;
  --typography-body-large-bold-font-family: 'Raleway', 'RalewayRoman-Bold';
  --typography-body-large-bold-font-size: 18px;
  --typography-body-large-bold-letter-spacing: 0px;
  --typography-body-large-bold-line-height: 135%;
  --typography-body-large-bold-font-weight: 700;
```

Kuva 10. Lopulliset CSS-muuttujat, jotka Style Dictionary -paketti määritelmäkoodin avulla generoi design tokeneista.

3.4 Semanttiset design tokenit

Arkkitehtuurisesti design tokeneita pystyy käsittelemään myös eritasoisilla arkkitehtuuriperiaatteilla, kuten kuvassa 11 on esitetty. Tässä mallissa tasoja on kolme: määritelmä, semanttinen käyttö ja komponenttitason käyttö. Ideana on määrätä tietyt arvot määritelmätasolla, kuten ”brändin pääväri” tai ”brändin toissijainen väri”. Tämän jälkeen semanttisella tasolla voidaan ajatella esimerkiksi brändin pääväri ”onnistumisen värinä” ja viitata semanttisen tason design token päävärin design tokeniin. Tällöin esimerkiksi kuvake, jonka tarkoitus on indikoida onnistunutta ostosuoritusta, voisi saada värikseen design tokenin ”icon-success” (vapaa suomennos ”kuvake-onnistunut”) -määritteen, joka on johdettu semanttisen tason onnistuneen värin design tokenista.



Kuva 11. Kolmitasoinen arkkitehtuurikaava (24).

Tällä keinoin jokainen komponentti ja sen tyylittelymääritelmä saavat ensimmäisen tason design tokenista johdetun komponenttikohtaisen design tokenin. Näin komponentti tai sen tyylittelymääritelmä pystytään tarvittaessa sitomaan eri määritelmätason design tokeniin, ilman että komponenttitason design tokenia tarvitsee jokaisessa komponentin ilmentymässä muuttaa. Kolmitasoinen arkkitehtuurimalli tuo joustavuutta ja skaalautuvuutta design tokenien käyttöön. (24.)

Konkreettisen hyödyn esimerkkinä on tilanne, jossa yrityksen ohjelmistossa on 150 ilmentymää määrätystä komponentista, jonka värimääritelmää pitää muuttaa brändin pääväristä toissijaiseen väriin. Sen sijaan, että ohjelmistokehittäjän täytyisi muuttaa 150 komponentin väriarvo pääväristä toissijaiseen, hän voi muuttaa kyseisen komponentin tietyn design tokenin viittauksen pääväristä toissijaiseen väriin ja säästää merkittävän määrän aikaa.

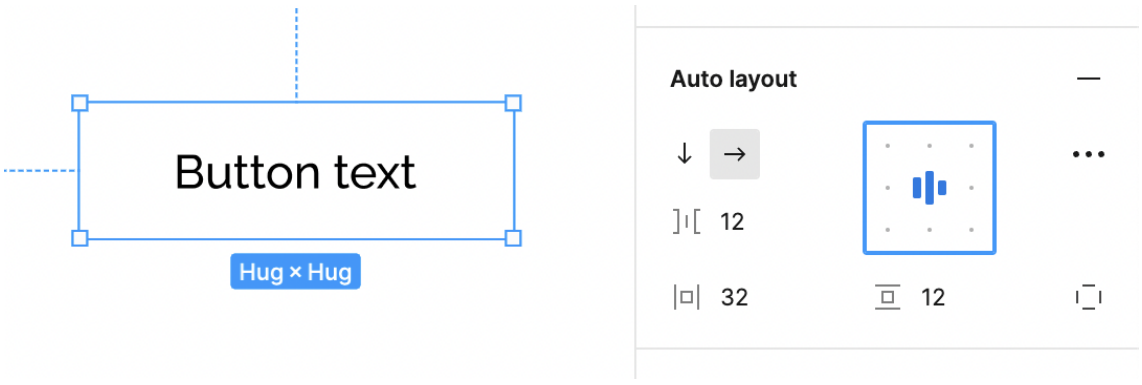
4 Uudelleenkäytettävät komponentit

4.1 Jaettavan komponentin suunnittelu

Design systemien ytimen voidaan nähdä rakentuvan komponenteista ja komponentit rakentuvat tyyleistä. Komponentit ovat käyttöliittymän osia, jotka kuvastavat tiettyä interaktiota tai muuta käyttöliittymätarvetta (25). Uudelleenkäyttämällä mahdollisimman montaa eri komponenttia, säästyy aikaa niin suunnittelijoilta kuin ohjelmistokehittäjiltä. Komponenttien ylläpitäminen design systemissä vähentää myös päällekkäisen työn riskiä eri tiimien välillä sekä mahdollistaa nopeiden prototyyppien luomisen. (26.)

Yleisimpiä komponentteja ovat esimerkiksi erilaiset painikkeet, kuvakkeet ja kentät. Prototyyppien ja muiden suunnitelmien toteuttaminen komponenteilla takaa sen, että komponentit ovat yhdenmukaisia ja ne ovat myös saatavilla muille tiimeille (27). Komponenttien rakentamiseen on suositeltavaa käyttää sekä Figman variant- ja component properties- että auto layout -toimintoja, joiden avulla seuraava demonstraatio painikekomponentin luomisesta on tehty.

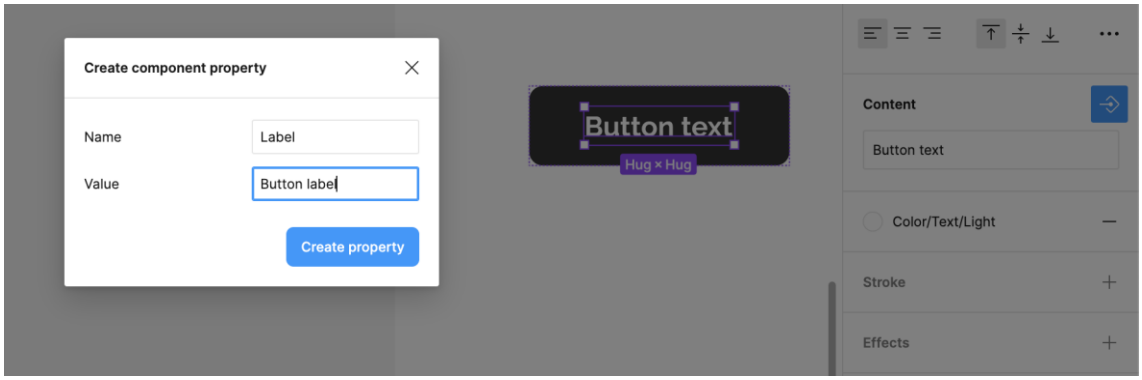
Komponentin voi luoda Figmassa mistä tahansa, ja tämän insinöörityön design systemissä luotiin painike tekstikentästä. Tekstikentästä tehtiin frame eli kehys, jotta siihen pystyi lisäämään Figman auto layout -toiminnon. Auto layout on hyvä kohdentaa keskelle, sillä painikkeen teksti ja mahdolliset kuvakkeet painikkeen sisällä on suositeltavaa sijoittaa keskelle, vaikka painikkeen leveys muuttuisi. Auto layout -toimintoon syötetään halutut pikseliarvot reunojen ja framen sisäisten elementtien välitykselle (kuva 12).



Kuva 12. Kehystetty tekstikerros, josta rakentuu painikekomponentti.

Kun painikkeen koko ja välitykset olivat halutunlaiset, voitiin painike tyylitellä. Esimerkissä lisättiin design systemin päävärin oletusarvon painikkeen taustaväriksi, web-kontrastivaatimusten takaavan tekstivärin ja reunojen pyöristystä. Tämän jälkeen painikekomponentin oletusvariantti oli valmis ja framesta voitiin tehdä Figma-komponentti valitsemalla ”Create component”.

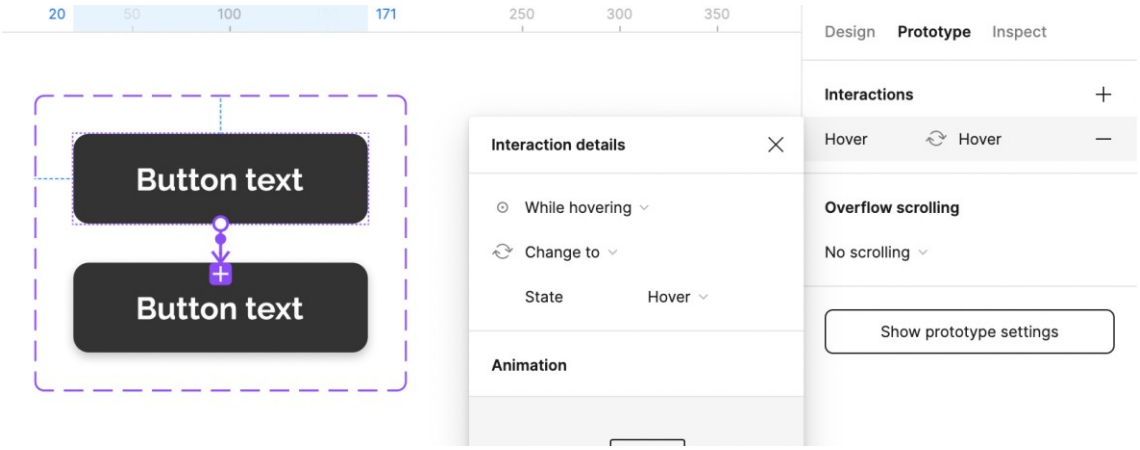
Jotta komponentit olisivat mahdollisimman helposti uudelleenkäytettäviä suunnittelijoiden näkökulmasta, Figmassa pystyy luomaan component properties- eli komponenttiominaisuuksia. Tämä tarkoittaa, että komponentilla on erilaisia ominaisuuksia, joita suunnittelija pystyy konfiguroimaan suoraan Figman ohjauspaneelista valitsemalla haluamansa komponentin. Ominaisuuksia voivat olla esimerkiksi kuvakkeen näyttäminen komponentissa, komponentin eri tilat, komponentin eri koot ja komponentin tekstien arvot (kuva 13).



Kuva 13. Komponentin tekstiominaisuuden luominen. Nimellä ”Label” viitataan painikekomponentin tekstiin ja sen oletusarvoksi annetaan ”Button label”.

Kun painikkeen ominaisuudet ja oletustyylit on määritelty, voidaan komponentista luoda eri ilmenemiä eli variantteja. Näillä tarkoitetaan esimerkiksi sitä, miten painike reagoi käyttäjän hiiren painallukseen ja mahdolliset virhetilanteet näkyvät painikkeessa. Esimerkkinä on kuvassa 14 luotu variantti siihen, miten painike reagoi käyttäjän hiiren osuessa komponenttiin, eli niin kutsuttu ”hover state”. Komponentista luodaan variantti, jolle annetaan

ominaisuus, joka määrittää komponentin tilaa eli State. Tilan nimeksi tässä tapauksessa annetaan Hover. Figmassa pystyy rakentamaan interaktioita jaettavien komponenttien välille ”Prototype”-välilehdellä. Nämä interaktiot kulkeutuvat julkaisun mukana jokaiseen instanssiin. Tässä esimerkissä lisättiin ”While hovering” -interaktio, joka muuttaa komponentin State-ominaisuuden Hover-varianttiin.



Kuva 14. Interaktion rakentaminen prototyyppivälilehdellä.

4.2 Storybook komponenttien kotina

Komponentit ovat modernin frontend-kehityksen, eli ohjelmoinnin käyttäjälle näkyvän osion kehityksen, peruspilareita ja käyttöliittymät rakentuvat usein erinäisistä komponenttikokoelmista. Yleensä organisaatio, joka tilaa tai rakentaa itse design systemin, tuottaa useampaa kuin yhtä palvelua, jolloin skaalautuvuus ja niin suunnittelumateriaalien kuin koodattujen resurssien uudelleenkäytettävyys on avainasemassa.

Tässä insinöörityössä haluttiin löytää parhaita ohjelmistoja ja käytäntöjä design systemin resurssien teknistä jakamista varten. Komponenttien suunnittelu toteutettiin Figmassa ja dokumentointi Zeroheightissä. Seuraava askel oli saada näistä suunnitelluista komponenteista ohjelmoidut versiot omaan kokonaisuuteensa, joita pystyttäisiin helposti ylläpitämään, kehittämään ja jakamaan kirjastona. Tähän tarkoitukseen valittiin design systemin komponenttikirjaston luomista varten palvelu nimeltä Storybook.

Storybook tarjoaa frontend-ohjelmoijien työhön paljon helpotusta ja uudelleenkäytettävyyttä. Monet yritykset rakentavat yrityksen laajuisen Storybookin, jossa eri kehitystiimit voivat kehittää, testata ja jatkokehittää komponentteja frontend-kehitystä varten. Esimerkiksi kun painikekomponentti on kerran kehitetty Storybookiin, ei jokaisen kehitystiimin tarvitse sitä rakentaa alusta alkaen. Komponenttien rakentaminen ja testaaminen Storybookissa mahdollistaa komponenttien renderöimisen ilman erillisen sovelluksen käynnistämistä. Näin ohjelmistokehittäjä voi tutkia, miten koodattu komponentti käyttäytyy esimerkiksi alkuperäistä pidemmän tekstielementin kanssa ja tehdä tarvittavat toimenpiteet, jos komponentti rikkoutuu pidemmän tekstisyötteen takia. (28.)

Storybook koostuu storyistä, jotka ovat näytteitä komponentin renderöidystä tilasta (29). Stories-tiedosto koostuu yleensä monista eri storyistä, jotka ovat ohjelmistokehittäjälle mielenkiintoisia.

4.3 Storybookin luominen design systemille

Kuten luvun 3 esimerkissä mainittiin, design tokeneiden muuntamista ja design systemin komponenttikirjastoa varten luotiin yhteinen kansio. Seuraavaksi luotava Storybook tulee olemaan design systemin komponenttikirjasto, ja sitä varten luotiin uusi npm-projekti samaan kansioon design token -ohjelmakoodin kanssa aikaisemmin käytetyllä komentorivikomennolla ’npm init -y’.

Modernissa web-kehityksessä on yleistä käyttää React-nimistä viitekehystä, joka vastaa komponenttien ja muiden käyttäjälle näkyvien osien dynaamisestä renderöinnistä (30). Koska komponentit tulivat olemaan React-pohjaisia, lisättiin myös projektiin tarvittavat React-riippuvuudet komennolla ’npm i react react-dom’. Koska näitä komponentteja tultiin jakamaan eteenpäin, tuli projektissa myös varmistaa niiden toimivuus projektin ulkopuolella. Npm-projekteissa tätä varten kehitetty peerDependency- eli vertaisriippuvuusominaisuus, jolla jaettu paketti varmistaa, että käyttäjäprojektilla on määrätyt riippuvuudet. Siksi React-riippuvuudet lisättiin vertaisriippuvuuksiksi package.json-tiedostoon. (31.)

Jotta tuotetut komponentit olisivat mahdollisimman toimivia, ymmärrettäviä ja ylläpidettäviä, käytettiin komponenttien luomisessa JavaScriptiin perustuvaa Microsoftin avoimen lähdekoodin ohjelmointikieltä TypeScriptiä. Se lisää JavaScriptin päälle tyypityksen, joka ilmoittaa tyyppivirheistä koodin rakentamisen aikana ja täten vähentää mahdollisten virheiden syntymistä koodin ajon aikana. Tyypitykset ovat erittäin tärkeitä uudelleenkäytettävissä komponenteissa, jottei esimerkiksi painikkeen tapahtumankäsittelijään syötetä vääränmuotoista tietoa, joka voisi pahimmassa tapauksessa rikkoa koko sovelluksen (32). TypesScriptin vaatimat riippuvuudet asennettiin komennolla ’npm i typescript @types/react’ ja lisättiin ohjelman juureen ’tsconfig.json’-niminen JSON-tiedosto, jonne määriteltiin TypeScriptin ominaisuuksia esimerkkikoodin 3 mukaan. Luvussa 3 generoitiin ohjelman juureen luotu ’variables.css’-tiedosto, jotta komponenttien käytössä olisivat suoraan design tokeneista johdetut tyylimuuttujat eikä jokaisen projektin tarvitsisi lisätä kyseistä tyylittelytiedostoa itse.

```
{
  "compilerOptions": {
    "jsx": "react",
    "target": "es2016",
    "outDir": "dist",
    "allowJs": true,
    "skipLibCheck": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "ES2020",
    "allowSyntheticDefaultImports": true,
    "moduleResolution": "Node",
    "declaration": true,
    "declarationDir": "dist"
  },
  "include": [
    "stories"
  ]
}
```

Esimerkkikoodi 3. tsconfig.json-määritelmäkoodi (33).

Näiden vaiheiden jälkeen ohjelma oli valmis Storybookin alustamiseen komennolla ’npx storybook init’. Tämä alustus tunnistaa automaattisesti käytettävän tekniikan TypeScriptiksi ja lisää projektiin muutaman esimerkkikomponentin TypeScript-tiedoston, tyylittelytiedostot ja ’stories.ts’-

tiedoston. Näiden lisäksi alustus lisää Storybookin vaatimat riippuvuudet ja tarvittavat skriptitiedostot. Oletusarvoisen Storybookin voi ajaa komennolla ’npm run storybook’. (34.)

4.4 Uudelleenkäytettävän React-komponentin luominen Storybookiin

Yksi yleisimmin käytetyistä komponenteista eri sovelluksissa on painikekomponentti, joten sitä käytetään esimerkkinä jakelukokonaisuuden lopputuotteena tässä insinöörityössä. Design systemin painikekomponentin luominen aloitettiin muokkaamalla Storybookin alustuksen mukana tullutta Button.ts-tiedostoa. Tiedosto nimettiin uudelleen DSButton.ts-nimiseksi kuvastamaan komponentin olevan design systemin painike. Tämän jälkeen tiedostoon lisättiin koodirivi ’import ’./variables.css’’, jolla otettiin käyttöön aikaisemmin tämän kansion rakenteeseen design tokeneista generoitu tyylitiedosto.

Tässä insinöörityössä keskityttiin toteuttamaan painikekomponentin design tokeneista johdetuilla muuttujilla, mikä vastaa visuaalisesti alkuperäisen Figma-suunnittelutiedoston suunnitelmaa painikekomponentista. Painikekomponentti tarvitsi alkuperäisen suunnitelman mukaan variantit primary, secondary ja tertiary, koot large, medium ja small ja eri tilat jokaiselle variantille, kuten hover, focused ja active.

Painikekomponentille luotiin TypeScriptin liittymä DSButtonProps kuvan 15 mukaan. Liittymä määrittelee ominaisuudet ja ominaisuuksien tyypit, joita painikekomponentti vastaanottaa. Näin pystyttiin varmistamaan, että komponentin käyttäminen on turvallista, sillä vääränmuotoisen tiedon syöttö aiheuttaa virheen koodin rakennuksen aikana (35). Liittymä myös kertoo implisiittisesti ohjelmistokehittäjälle, mitä komponentille tulee syöttää ja mitä eri ominaisuuksia painikkeella on.

```
export interface DSButtonProps {
  variant: string,
  size: 'small' | 'medium' | 'large';
  label: string,
  disabled: boolean,
  onClick: () => void,
}
```

Kuva 15. DSButtonProps-liittymä, jolla varmistetaan syötetyn tiedon haluttu muoto.

Näiden ominaisuuksien avulla ohjelmistokehittäjä pystyy lähdekoodissa (kuva 16) konfiguroimaan, miltä komponentin tulee näyttää ja mikä sen toiminnallisuus on.

```
<div className='button-container'>
  <div className='button-row'>
    <Button variant={'primary'} label={'Fetch tokens'} size={'large'} />
    <Button variant={'primary'} label={'Fetch tokens'} size={'medium'} />
    <Button variant={'primary'} label={'Fetch tokens'} size={'small'} />
  </div>
```

Kuva 16. Painikekomponentin käyttö lähdekoodissa.

Jotta painikekomponentti osaa hyödyntää lähdekoodissa syötettyjä ominaisuuksia ulkoasunsa määrittelemiseen, lisättiin painikkeen luokka-attribuuteiksi syötettyjä arvoja (kuva 17). Näin esimerkiksi lähdekoodissa syötetty ”large” eli kokoa kuvaava merkkijono-ominaisuus määrittelee painikkeen luokka-attribuutiksi ’ds-button—large’, jolloin se saa nimeä vastaavat tyylittelyarvot ’DSButton.css’-tiedostosta. Tyylittelytiedostossa määriteltiin jokainen variantti, koko ja komponentin tila omina luokkinaan design tokeneista johdetuilla CSS-muuttujilla, joita komponentin lähdekoodi määrittelee.

```
export const DSButton = ({
  variant,
  size,
  label,
  disabled,
  ...props
}: DSButtonProps) => {
  const sizeClass = `ds-button--${size}`
  const variantClass = `ds-button--${variant}`
  return (
    <button
      type='button'
      className={['ds-button', sizeClass, variantClass].join(' ')}
      disabled={disabled}
      {...props}
    >
      {label}
    </button>
  )
}
```

Kuva 17. Painikekomponentin koodi, joka käsittelee ohjelmistokehittäjän syöttämän tiedon perusteella oman ulkoasunsa.

Generoimalla aikaisemman esimerkin mukaan CSS-muuttujat design tokenien perusteella mahdollistettiin suunnittelijan määrittelemien tyyliarvojen käytön suoraan ohjelmoinnissa. Tehtäessä painikekomponentin tyylittelyä ei ohjelmistokehittäjän tarvitse miettiä, mikä esimerkiksi brändin käyttämän päävärin heksadesimaaliväriarvo on, sillä hän voi suoraan käyttää generoituja nimettyjä muuttujia. Koska muuttujat johdetaan suoraan Figman tyyleistä, on ohjelmistokehittäjän helppo katsoa, minkäniminen tyyli on käytössä esimerkiksi painikkeen taustavärinä, ja tämän perusteella määritellä kyseisen tyylin nimestä johdettu CSS-muuttuja koodatun komponentin taustaväriksi.

Ohjelmistokehittäjä pystyy myös tekemään tarvittaessa eritasoisia määritelmiä muuttujien kanssa, eritasoisten design tokeneiden esimerkin mukaisesti. On esimerkiksi mahdollista, että ’DSButton.css’-tyylittelytiedostoon määritellään omat muuttujat, kuten esimerkiksi ’--button-color-default’, joka viittaa tiettyyn brändin väriin. Tällä tavoin ohjelmistokehittäjä pystyy määrittelemään kyseisen komponentin osien värit kontrolloidummin. Jos suunnittelussa päätetään muuttaa, mitä väriä primäärin painikevariantin taustavärin ja sekundäärisen painikkeen reunojen värin tulee käyttää, voi ohjelmistokehittäjä yksinkertaisesti

vain muokata yhtä viittausta koodissaan, ilman että hänen tarvitsee muuttaa jokaista kyseisen värin ilmentymää tyylittelytiedostossa (kuva 18).

```
:root{
  --button-color-default: var(--color-main-main);
  --button-color-active: var(--color-main-dark);
  --button-color-inactive: var(--color-main-faded);
  --button-text-inactive: var(--color-text-main);
  --button-text-default: var(--color-text-light);
  --button-text-negative: var(--color-text-dark);
}
```

Kuva 18. Koodieditorissa luotuja muuttujaviittauksia, jolla komponentteja voi tyylitellä.

5 Design systemin jakaminen ja hyödyntäminen koodiympäristössä

5.1 Komponenttikirjaston jakaminen

Jotta komponenttikirjasto saadaan jaettua mahdollisimman helppokäyttöisenä kokonaisuutena organisaation jokaisen kehitystiimin kesken, on hyvä ajatus käyttää kehittäjille tuttuja teknologioita, kuten npm:ää. Npm on maailman suurin ohjelmistorekisteri, joka toimii avoimen lähdekoodin periaatteella ja jossa eri tahot pystyvät käyttämään ja jakamaan erinäisiä koodipohjaisia paketteja. (36.)

Jotta design systemin komponenttikirjasto saatiin julkaistua npm-pakettina, tarvittiin käyttäjä npm:ään. Jos design systemiä tehdään yritykselle, voi yritys tehdä oman käyttäjätilin npm:ään, jossa on tarjolla maksullinen ”Organisation”-suunnitelma, joka mahdollistaa yksityisten npm-pakettien julkaisun. Tällä tavoin yritykset voivat pitää pääsynsä omiin kehitysresursseihinsa rajattuna. (36.)

Storybookin jakamisen valmistelu aloitettiin lisäämällä ohjelman ”stories”-nimiseen kansioon ”index.ts”-tiedosto, joka toimii sovelluksen tulokohtana ja määrittelee, mitkä komponentit tullaan julkaisemaan design systemin npm-

paketissa. Jotta luotu painikekomponentti DSButton saatiin npm-pakettiin julkaistavaksi, tuotiin se index.ts-tiedoston käyttöön ja vietiin se ulos (kuva 19).

Kuva 19. Index.ts-tiedosto, jossa määritellään, mitkä komponentit halutaan sisällyttää julkaistavaan pakettiin.

Jotta kehitelty lähdekoodi saatiin julkaistavaan muotoon, tarvittiin moduulipaketoijaa, joka teki lähdekoodista jaettavan version projektiin. Rollup-kirjasto on tarkoitettu kirjastojen tai sovellusten julkaisemiseen, joten se on optimaalinen työkalu komponenttikirjaston julkaisemiseen (37). Rollup vaatii toimiakseen alustuksen ja useamman eri riippuvuuden määrittelyn terminaalikomennolla sekä oman määrittelytiedoston, jotka on listattu esimerkkikoodissa 4 ja 5.

```
npm i --save-dev rollup @rollup/plugin-node-resolve @rollup/plugin-ba-
bel rollup-plugin-uglify rollup-plugin-postcss rollup-plugin-type-
script2 rollup-plugin-peer-deps-external
```

Esimerkkikoodi 4. Rollup-alustuksen ja riippuvuuksien määrittelyn komentorivikäsky.

```
{
  "compilerOptions": {
    "jsx": "react",
    "target": "es2016",
    "outDir": "dist",
    "allowJs": true,
    "skipLibCheck": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "ES2020",
    "allowSyntheticDefaultImports": true,
    "moduleResolution": "Node",
    "declaration": true,
    "declarationDir": "dist"
  },
  "include": [
    "stories"
  ]
}
```

Esimerkkikoodi 5. Rollup-määrittelytiedosto.

Jotta kehitetty ohjelma osaa hyödyntää Rollupia ja luoda jaettavan version koodista, tulee Rollup suorittaa omalla komentorivikomennolla. Tämän komennon voi lisätä ohjelman package.json-tiedoston skripteihin ’build-ds": "rollup -c –bundleConfigAsCjs’. Saman package.json-tiedoston tulokohta sovellukseen tulee muuttaa muotoon ”dist/main.js”, jotta julkaistu paketti käyttää Rollupilla generoitua versiota (kuva 20).

```
{
  "name": "@npm-user/ts-storybook-components",
  "version": "1.0.2",
  "description": "TS components for Design system",
  "main": "dist/index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "storybook": "start-storybook -p 6006",
    "build-storybook": "build-storybook",
    "build-ds": "rollup -c --bundleConfigAsCjs"
  },
```

Kuva 20. Muokattu package.json-tiedosto paketin julkaisua varten.

Ennen julkaisua package.jsonissa voi määritellä julkaistavan paketin nimen ja versioinnin. Nimeksi tulee määritellä npm:n rekisterissä uniikki nimi ja versio,

jotta julkaisu onnistuu. Määrittelemällä uusia julkaisuversioita pystyy organisaatio hallitsemaan muutoksia tulevaisuudessa.

Tämän jälkeen komponenttikirjasto oli valmis julkaistavaksi. Npm-paketin julkaisu on yksinkertaista: käyttäjän tarvitsee vain kirjautua sisään komentorivikäskyllä ’npm login’ ja suorittaa julkaisukäsky ’npm publish’. Tämän jälkeen package.json-tiedoston mukainen paketti ja versio on vapaasti käytettävissä npm:n kautta.

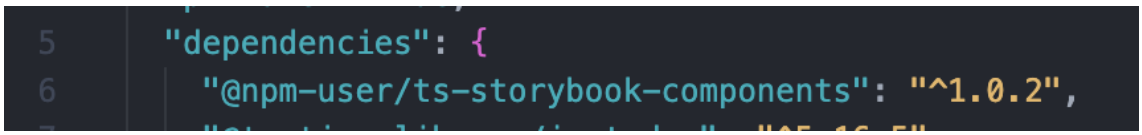
5.2 Julkaistun kirjaston käyttäminen ja muutosten versiointi

Kun komponenttikirjastosta generoitu paketti on julkaistu, sitä voi käyttää kuka tahansa npm:n kautta. Riittää, että projektissa ajetaan komento ’npm i <julkaistuun paketin nimi>’, jolloin julkaistu paketti asentuu riippuvuudeksi projektiin. Tämän jälkeen paketista pystyy tuomaan komponentteja käyttöönsä; esimerkiksi julkaistun painikekomponentin saa käyttöönsä esimerkkikoodilla 6.

```
import { DSButton } from '@<npm-user>/ts-storybook-components'
```

Esimerkkikoodi 6. Painikekomponentin tuonti React-projektin käyttöön.

Paketti näkyy riippuvuutena package.json-tiedostossa, josta myös näkee paketin versioinnin. Versioinnilla on helppo kommunikoida muutoksia laajemmin, ja design systemissä on hyvä määrätä tarkemmat ohjeet versioinnille. Yksi yleinen tapa on käyttää versiointiin mallia kuten kuvassa 21 on käytetty, jossa ensimmäinen numero on laajoille muutoksille, toinen pienemmille muutoksille ja kolmas numero pienille paranteluille (38).



Kuva 21. Design systemistä julkaistu npm-paketti riippuvuutena projektissa.

Uuden version tekeminen design systemistä vastaavalla tiimillä on suoraviivaista. Kun suunnittelijoiden tekemät tyylien muutokset on suoritettu Figmassa, tulee Zeroheightin lataus päivittää, jolloin design tokenien arvot dokumentoinnissa muuttuvat. Tämän jälkeen riittää, että suorittaa design tokeneiden muuntamiseen käytetyn ohjelmakoodin uudelleen, sillä ohjelmassa on käytetty Zeroheightin url-osoitetta, jossa ovat nyt päivittyneet design tokenit. Ohjelman suorittamisen jälkeen uudet päivitetyt tyylit ovat suoraan komponenttikirjastossa, sillä esimerkissä määritettiin tyylitiedoston generoituvan suoraan komponenttikirjaston juureen. Riippuen muutosten laajuudesta tulee versiointi päivittää design systemin ohjeiden mukaisesti komponenttikirjaston koodissa, minkä jälkeen komponenttikirjastosta tulee julkaista uusi paketti.

Tämän jälkeen design systemistä vastaava tiimi kommunikoi uuden version olevan käytettävissä, ja kehitystiimi voi helposti käydä päivittämässä versionumeron riippuvuuksista ja suorittaa ”npm i”-komennolla projektin riippuvuuksien asennuksen. Muuttuneet arvot ovat käytössä automaattisesti ohjelmistoprojektissa. Näin suunnittelumuutosten tuominen koodiprojektiin ei vaadi ohjelmistokehittäjältä muuta kuin riippuvuuksien päivityksen, sen sijaan että ohjelmistokehittäjän pitäisi muuttaa lukuisia eri arvoja tyylitiedostosta.

6 Yhteenveto

Insinöörityössä perehdyttiin erinäisten tekniikoiden ja ohjelmistojen kautta design systemin tekniseen jakeluun ja dokumentoitiin siihen liittyvät parhaat käytännöt mahdollisimman yleiskäyttöisen kokonaisuuden rakentamiseksi ja ylläpitämiseksi. Työ osoitti design systemin teknisen jakelun olevan moniosainen kokonaisuus, mutta erittäin oleellinen tapauksessa, jossa useat kehitystiimit tarvitsevat keskenään samoja resursseja.

Työssä luotiin Figma-suunnittelukirjasto, josta dokumentointityökalulla Zeroheight saatiin generoitua kehittäjien tarvitsemat design tokenit. Nämä design tokenit muunnettiin web-kehittäjien tarvitsemaan muotoon ja rakennettiin Storybook komponenttien kehitystä ja testausta varten. Storybook jaettiin npm-

pakettirekisterin kautta, josta kehitystiimit pystyvät helposti hyödyntämään design systemin komponentteja.

Kun design systemin luomisessa alusta asti seurataan mahdollisimman kauaskantoista lähestymistapaa, saadaan rakennettua tekninen jakelukokonaisuus, jossa suunnittelumuutokset johdetaan helposti suoraan ohjelmistoon. Päivittämällä Zeroheightin dokumentaation Figma-resurssi ja suorittamalla design tokenit muuntava ohjelmisto, on komponenttikirjastossa suoraan käytössä ajantasaiset tyyliresurssit. Tällöin ohjelmointitiimi pystyy kontrolloidusti päivittämään ohjelman riippuvuudeksi asetetun komponenttikirjaston versiointinumeron, jolloin suunnittelijan luomat muutokset näkyvät lopputuotteessa välittömästi.

Lähteet

1 Hacq, Audrey. 2018. Everything you need to know about Design Systems. Verkkoaineisto. UX Collective. <https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969>. Luettu 13.2.2023.

2 The Guardian Design system. Verkkoaineisto. The Guardian. <https://design.theguardian.com/>. Luettu 27.3.2023.

3 Federica, Bruno. 2022. 2022 is the year of Design Systems. Verkkoaineisto. Bootcamp. <https://bootcamp.uxdesign.cc/2022-is-the-year-of-design-systems-40233987cb45>. Luettu 1.3.2023.

4 Design systems survey. 2022. Verkkoaineisto. Sparkbox. <https://designsystemsurvey.sparkbox.com/2022/>. Luettu 1.3.2023.

5 Brown, Rachel. Why Should You Build an Enterprise Design System? 5 Benefits. Verkkoaineisto. Ionic. <https://ionic.io/resources/articles/why-every-company-needs-a-design-system>. Luettu 6.2.2023.

6 Bakir, Molham. The Benefits of a Design System: Making Better Products, Faster. Verkkoaineisto. Toptal. <https://www.toptal.com/designers/design-systems/benefits-of-design-system>. Luettu 1.3.2023.

7 Gavinet, Fabien. 2022. Colors that make sense. Verkkoaineisto. Medium. <https://medium.com/getaround-eu/colors-that-make-sense-505d0f3509c1>. Luettu 28.3.2023.

8 Style Dictionary. Verkkoaineisto. Amazon. <https://amzn.github.io/style-dictionary/#/README>. Luettu 28.3.2023.

9 Brett, Will. 2022. 8 design system management tools for startups & organizations. Verkkoaineisto. Anima. <https://www.animaapp.com/blog/industry/design-system-tools-for-organizations/>. Luettu 6.2.2023.

10 Design System in Figma. Verkkoaineisto. Design+Code. <https://design-code.io/design-system-in-figma>. Luettu 6.2.2023.

11 Styles in Figma. Verkkoaineisto. Figma. <https://help.figma.com/hc/en-us/articles/360039238753-Styles-in-Figma>. Luettu 20.2.2023.

12 Collaboration is hard. We make it easier. Verkkoaineisto. Figma. <https://www.figma.com/collaboration/>. Luettu 7.2.2023.

13 Andrew, Marc. 2021. Creating a design system in Figma: a practical guide. Verkkoaineisto. UX Collective. <https://uxdesign.cc/creating-a-design-system-in-figma-cbd01b0d2424/>. Luettu 1.3.2023.

14 Zeroheight-verkkosivusto. Verkkoaineisto. Zeroheight. <https://zeroheight.com/>. Luettu 1.3.2023.

15 Mahe, Jules. 2022. How to document your design system components. Verkkoaineisto. Zeroheight. <https://zeroheight.com/blog/how-to-document-your-design-system-components/>. Luettu 6.2.2023.

16 Morales, Justin. 2021. What Are Design Tokens? A Design Systems Tool. Verkkoaineisto. Adobe. <https://xd.adobe.com/ideas/principles/design-systems/what-are-design-tokens/>. Luettu 27.2.2023.

17 Design tokens. Verkkoaineisto. Adobe. <https://spectrum.adobe.com/page/design-tokens/>. Luettu 13.3.2023.

18 Mahe, Jules. 2022. Design tokens as your DNA. Verkkoaineisto. Zeroheight. <https://zeroheight.com/blog/design-tokens-as-your-dna/>. Luettu 27.2.2023.

19 Design Tokens: How to use them effectively. 2020. Verkkoaineisto. UX Collective. <https://uxdesign.cc/design-tokens-how-to-use-them-effectively-d495ff05cbbf>. Luettu 27.2.2023.

20 Gonzalez, Oscar. 2021. Design tokens cheatsheet. Verkkoaineisto. UX Collective. <https://uxdesign.cc/design-tokens-cheatsheet-927fc1404099>. Luettu 13.3.2023.

21 Npm-init dokumentaatio. Verkkoaineisto. Npm. <https://docs.npmjs.com/cli/v8/commands/npm-init>. Luettu 28.2.2023.

22 Style Dictionary konfiguraatiosivusto. Verkkoaineisto. Amazon. <https://amzn.github.io/style-dictionary/#/config>. Luettu 27.2.2023.

23 CSS Tutorial. Verkkoaineisto. W3Schools. <https://www.w3schools.com/css/>. Luettu 27.2.2023.

24 Frost, Brad. 2022. The many faces of themeable design systems. Verkkoaineisto <https://bradfrost.com/blog/post/the-many-faces-of-themeable-design-systems>. Luettu 27.2.2023.

25 Atlassian Design system. Verkkoaineisto. Atlassian. <https://atlassian.design/components/>. Luettu 13.2.2023.

26 Lindberg, Oliver. 2020. The Main Pillars of Effective Design Systems. Verkkoaineisto. Adobe. <https://xd.adobe.com/ideas/principles/design-systems/main-pillars-effective-design-systems/>. Luettu 13.2.2023.

27 Guide to components in Figma. Verkkoaineisto. Figma. <https://help.figma.com/hc/en-us/articles/360038662654-Guide-to-components-in-Figma/>. Luettu 13.2.2023.

28 Hawkins, Tyler. 2020. Why You Should Always Use Storybook When Developing User Interfaces. Verkkoaineisto. Level Up Coding. <https://levelup.gitconnected.com/why-you-should-always-use-storybook-when-developing-user-interfaces-4c69b93b2f65>. Luettu 28.2.2023.

29 What's a Story. Verkkoaineisto. Storybook. <https://storybook.js.org/docs/react/get-started/whats-a-story>. Luettu 28.2.2023.

30 Peer Dependencies. Verkkoaineisto. The OpenJS Foundation. <https://nodejs.org/es/blog/npm/peer-dependencies/>. Luettu 28.2.2023.

31 Herbert, David. 2022. What is React.js? (Uses, Examples, & More). Verkkoaineisto. Hubspot. <https://blog.hubspot.com/website/react-js/>. Luettu 28.2.2023.

32 TypeScript for JavaScript Programmers. Verkkoaineisto. Microsoft. <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html/>. Luettu 28.2.2023.

33 Rao, Akilesh. 2020. NPM Tutorial (2021) Part 3 : Create and publish your own React component library. Verkkoaineisto <https://www.youtube.com/watch?v=d8oztfRBGrI>. Luettu 28.2.2023.

34 Install Storybook. Verkkoaineisto. Storybook. <https://storybook.js.org/docs/react/get-started/install>. Luettu 28.2.2023.

35 Masand, Ankita. 2022. Understanding and using interfaces in TypeScript. Verkkoaineisto. LogRocket <https://blog.logrocket.com/understanding-using-interfaces-typescript/>. Luettu 28.2.2023.

36 About npm. Verkkoaineisto. Npm. <https://docs.npmjs.com/about-npm>. Luettu 27.2.2023.

37 Rollup. Verkkoaineisto. Rollup. <https://rollupjs.org/introduction/>. Luettu 27.2.2023.

38 A Complete Guide to Understanding Software Version Numbers. Verkkoaineisto. Praxent. <https://praxent.com/blog/what-do-software-

version-numbers-mean-a-guide-to-understanding-software-version-conventions>. 27.2.2023.