Tero Tanttu

STORYBOOK OSANA KOMPONENTTILÄHTÖISTÄ OHJELMISTOKEHITYSTÄ

Tietojenkäsittely

2019

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Tero Tanttu	Tradenomi (AMK)	Marraskuu 2019

Opinnäytetyön nimi

Storybook osana komponenttilähtöistä ohjelmistokehitystä

51 sivua
5 liitesivua

Toimeksiantaja

Observis Oy

Ohjaaja

Janne Turunen

Tiivistelmä

Tämän opinnäytetyön päämääränä on tutkia komponenttilähtöisen kehityksen ominaisuuksia ohjelmistokehityksessä Storybook-työkalua käyttäen React -projektissa. Työ on osa toimeksiantajan tavoitetta löytää uusia toimivia työskentelytapoja ja työkaluja omaan komponenttikehityksensä tueksi, erityisesti heidän omaan ObSAS 2.0 -projektiin, joka on käyttöliittymäluontityökalu. Tämä nopeuttaisi uusien komponenttien kehitystyötä säilyttäen komponenttien eheyden ja mahdollistaisi niiden paremman dokumentoimisen ja uudelleenkäytön.

Opinnäytetyöraportin teoriaosuudessa käydään läpi komponenttilähtöisen kehityksen perusperiaatteita ja toimintotapoja sekä, mikä komponentti on ohjelmistokehityksessä. Tämän lisäksi käydään läpi Reactin ja sen eri komponenttien ominaisuuksia sekä tapoja, miten komponentteja voidaan kehittää ja missä tilanteissa eri kehitystavat toimivat esimerkein. Viimeisenä esitellään Storybook ja sen toiminnallisuus teoreettisesti. Raportin käytännön osuudessa esitellään Storybookin toiminta tarkemmin. Siinä rakennetaan ja asennetaan halutunlaisilla lisäominaisuuksia Storybook-työskentely-ympäristö ObSAS 2.0 -projektiin. Lopulta käydään läpi projektin yhden esimerkkikomponentin avulla Storybookin haluttua toiminnallisuutta edellä mainitussa projektissa.

Työn erityisenä haasteena on löytää tapa, jolla yhdistetään ObSAS 2.0 -projektin ja sen komponenttien, joilla on erityisen vahva riippuvuus Redux-ympäristöön, osaksi Storybook-työskentely-ympäristöä. Myös halutunlaisen jäljittelytiedon saaminen Storybook-kehitystä varten sekä kehittäjien vaihtelevat kehitystavat komponenttien kehityksen ovat tuottaneet ongelmia projektissa.

Lopputuloksena saadaan kohtuu hyvin toimeksiantajan tarpeisiin sopiva kehitysympäristö, joka täyttää annetut tavoitteet, vaikkakin täydellinen se ei ole. Se tuo esiin erityisesti ne ongelmakohdat, jotka vaativat vielä jatkokehitystä ja jotka on otettu tämän takia osaksi toimeksiantajan kehitystavoitteita.

Asiasanat

käyttöliittymä, ohjelmointi, dokumentointi, testaus, Storybook, TypeScript, React, komponenttilähtöinen kehitys

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Tero Tanttu	Bachelor of Business Administration	November 2019

Thesis title

Component-driven development with Storybook

51 pages
5 pages of appendices

Commissioned by

Observis Oy

Supervisor

Janne Turunen

Abstract

This thesis was commissioned by the Observis Oy company that is highly focused on improving its own software development capabilities, especially in React and component development. The objective of this thesis was to research Storybook features and how development tool can be included in software development for the ObSAS 2.0 React project which is a tool in development for creating user interfaces. These changes may have major impact improving component development and component reusability and documentation.

The theory of the bachelor's thesis examines the basics and methods of component-driven development and defined what is component in software development. In addition, it examined React and its various component properties and developmental methods to create components and in which situations different development methods worked through examples. Lastly the theory will go through Storybook and its functionality theoretically. The practical part of the bachelor's thesis showed how Storybook workspace was build and installed in the ObSAS 2.0 project. Finally, Storybook functionality is presented with one of the ObSAS 2.0 project's pre-made component.

The biggest problem in the practical part was finding a solution for how the Storybook and ObSAS 2.0 components that were heavily connected to the Redux environment would be attached to together. Also getting kind of mock data needed in Storybook, and different developers nonunified way to develop components caused problems.

The end resulted in a moderately well working workspace for Storybook, even though it was not perfect. It showed the company what parts of development needed to further development, and because of these results they will be consider the company's development objectives.

Keywords

user interface, programming, documentation, testing, Storybook, TypeScript, React, component-driven development

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön päämääränä on käydä läpi komponenttilähtöisen kehityksen ominaisuuksia ohjelmistokehityksessä. Samalla esittelen yhtä tämän kehityssuuntaa auttavaa-työkalua, Storybookia ja sen tarjoamia hyötyjä ja kuinka ne voidaan ottaa osaksi tätä ohjelmistokehitysprosessia Observis Oy:n ObSAS 2.0 -projektissa. Työympäristönä toimii React. Opinnäytetyö odottaa lukijalta jonkin asteista kokemusta ohjelmoinnista ja Reactista opinnäytetyön sisällön ymmärtämiseksi.

Työ on toteutettu toimeksiantona Observis Oy:lle, joka pyrkii jatkuvasti kehittämään omia työskentelytapoja yrityksen kasvun myötä automatisoidumpaan suuntaan. Tämän toimeksiannon myötä kehityssuunnan tarkoitus on siirtää työprosessia paljon yksityiskohtaisempaa ja insinöörimäisempiin työtapoihin. Päämääränä on suunnitella käyttöliittymät yksityiskohtaisesti ennen kuin yhtään koodiriviä on kirjoitettu ja auttamaan tätä pyrkimystä, tavoitteena on kehittää kattava React -komponenttikirjasto, jonka sisältö sopii yrityksen tuotteisiin. Samalla kehitystiimin on tarkoitus kehittää omaa osaamistaan enemmän komponenttien suunnittelussa ja kehityksessä. Edellä mainittujen asioiden vuoksi Observis Oy on kiinnostunut ottamaan käyttöön Storybook-työkalun. Tämän uskotaan auttavan saavuttamaan yrityksen tavoitteita sekä kehittämään jo olemassa olevia asioita eteenpäin, kuten komponenttien testausta sekä niiden dokumentointia.

Opinnäytetyön rakenne pyrkii luomaan ehjän kokonaisuuden aihepiiristä, alkaen komponenttilähtöisen kehityksen ajatusmaailmasta kohti Reactin esittelyä ja sen eri komponentteja sekä niiden eri kehitystapoihin. Viimeisenä teoriaosiossa käytävä Storybook-aihepiiri käy läpi Storybookin perustoiminnallisuutta, joka samalla pohjustaa käytännönosiota, jossa lopulta käydään läpi kehitysympäristön asennus ja konfigurointi esimerkein.

Esimerkkiprojektina käytännön osiossa toimii Observis Oy:n oma projekti, ObSAS 2.0, joka on työkalu käyttöliittymien luomiseen Observis Oy:n omille projekteille. Se on Electron-kehitysympäristössä toteutettu projekti, mikä mahdollistaa verkkosovellusmaisten sovellusten kehityksen työpöytäsovelluksina (Brockhoff 2018). Electron itsessään on yksittäinen ympäristö kehitykseen ja

ObSAS 2.0 kohdalla siihen on liitetty myös React- ja TypeScript-ympäristö, joissa varsinainen kehitystyö tapahtuu. Tämän vuoksi myös Storybook-työkalun asentaminen ja varsinainen komponenttien parissa työskentely projektin tavoitteiden vuoksi eroaa haastavuudessaan perinteisestä verkkosovelluskehityksestä paljonkin.

Tulen suurimmilta osin esittämään esimerkit pääasiassa TypeScript esimerkein. Esimerkkikuvat ovat järjestään englannin kielisiä johtuen yritykselle tehdystä toimeksiannosta, jossa englanti on pääsääntöinen työskentelykieli ja koska englanti on yleisesti ohjelmoinnissa käytetty pääkieli.

2 KOMPONENTTILÄHTÖINEN KEHITYS

Ongelma, joka yleensä tulee vastaan projekteissa sekä erityisesti suurissa projekteissa on tapa, kuinka yhdistetään tehokkuus, suunnittelu ja käytäntö. Usein ilman selkeää näkemystä komponentit ja niiden toiminnot ja ulkoasut suunnitellaan yhtä tilannetta ajatellen. Tästä seuraa se, että luodun komponentin uudelleenkäytettävyys kärsii tai sitä ei ole ollenkaan. Voi olla, että suunnittelussa ei oteta jotain tiettyä asiaa huomioon tai sen toimintoja lisätään kerta kerralta enemmän, jolloin komponentti ja sen toiminnallisuus turpoaa. Tästä alkaa seurata ongelmia pitkällä aikavälillä. Ennen siirtymistä varsinaisiin komponentteihin ja niiden suunnitteluun käytännössä, on hyvä tiedostaa mitä ajatus komponenttilähtöisestä kehityksestä (Component driven development) tarkoittaa ja miten se pyrkii vastamaan näihin edellä mainittuihin ongelmiin.

Ensimmäisiä ajatuksia komponenttipohjaisesta kehityksestä (component-based development) ja ohjelmistojen rakentamisesta valmiiden komponenttien avulla esiin toi Douglas McIlroy vuonna 1968. Tämä tapahtui ohjelmistokehittäjien Nato konferenssissa Saksan Garmisch kaupungissa. Hän on kehittänyt tämän ajatusmaailman mukaisesti muutamia Unix työkaluja, kuten sort- ja join- toiminnot. (Naur & Randell 1969, 79.)

Varsinaisen nykyaikaisen konseption ohjelmistokomponentille määritteli Brad Cox (Cox 1986). Hän myös kehitti Tom Loven kanssa tätä ajatusmaailmaa mukailevan Objective-C -olio-ohjelmointikielen 1980-luvun alkupuoliskolla, jota suurista yrityksistä on käyttänyt Apple (Hsu 2017).

Mutta mikä sitten on komponentti ohjelmistokehityksessä? Margaret Rouse (2016) määrittelee komponentin olevan tunnistettava osa suurempaa ohjelmaa tai rakennetta. Se tarjoaa toiminnon tai toimintojen joukon. Se on uudelleen käytettävä ja sitä voidaan käyttää muiden komponenttien kanssa. Se voi olla mitä tahansa graafisesta elementistä tai toiminnosta rajapinnan määrittelyyn. Tämä opinnäytetyö käsittelee komponentteja nimenomaan osana käyttöliittymää eli käyttöliittymäkomponentteina.

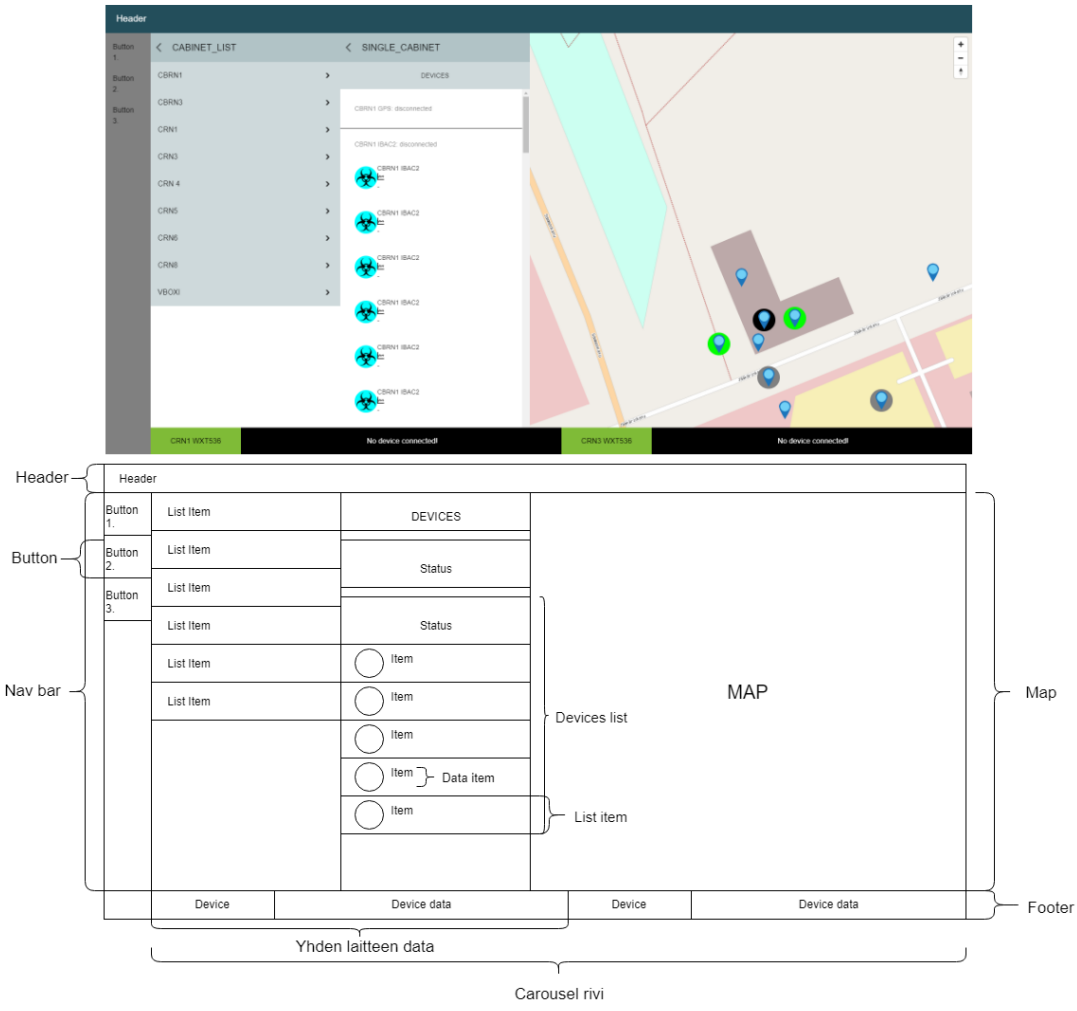
Ajatus komponenteista ja niiden mahdollisuuksista selainpuolen (front-end) kehityksessä ei ole myöskään uusi asia. Jo useita vuosia kehittäjät ovat kehittäneet modularisointi (modularization) ajatusmaailman mukaan sovelluksia ja sovellusten osia. Päämääränä luoda uudelleen käytettäviä, helposti muokkautuvia, ylläpidettäviä ja skaalautuvia eri tilanteisiin sopivia moduuleja. Esimerkkejä tästä voidaan huomata monessakin paikassa kuten tyylikirjastojen *grid* -järjestelmissä kun jaetaan tilaa eri käyttöliittymän osien kesken tai toimintoja niputetaan omiin pienempiin kokonaisuuksiin. (De La Cuadra 2016.) Samanlaista periaatetta kuin modularisointi, myös komponenttilähtöisesti voidaan suunnitella ja kehittää verkkosovelluksia.

Päämääränä kuten modularisoinnissa, niin myös komponenttilähtöisessä kehityksessä on tarkoitus luoda pienistä osista isoja kokonaisuuksia kuitenkin niin, että nämä pienet osat olisivat edelleen käytettävissä muuallakin. Avainsanat tähän ovat koheesio ja kytkentä.

Koheesiolla (cohesion) tarkoitetaan, kuinka hyvin tietty ohjelmakoodi on keskittynyt tietyn toiminnallisuuden toteuttamiseen. Käytännössä ohjelmakoodin koheesiota voidaan arvioida käsitteillä *korkea koheesio* ja *matala koheesio.* Yksinkertaistettuna korkea koheesio tarkoittaa parempaa uudelleen käytettävyyttä, luettavuutta ja ymmärrettävyyttä. Matala koheesio puolestaan päinvastaista. Koheesion tasoon vaikuttaa kytkennän (coupling) taso. Kytkennellä (coupling) tarkoitetaan riippuvuutta ohjelmamoduulien välillä, miten tiukasti toiminnot tai moduulit ovat riippuvaisia toisistaan. Esimerkiksi miten laajalle alueelle toiminnallisuus on jaettu ja kuinka riippuvainen moduuli on muista moduuleista (GeeksforGeeks 2019a).

2.1 Toimintaperiaate

Varsinainen kehitys alkaa komponenttilähtöisen kehitys -maailman mukaan pohjalta eli komponentit, näkymät ja valmiit sivut rakennetaan pohjalta ylöspäin, aina pienimmästä komponentista suurempaan (kuva 1) hyvin dokumentoituihin kokonaisuuksiin, kunnes on valmiita toiminnallisia kokonaisuuksia tai näkymiä/sivuja. Jokaisen yksittäinen komponentin voidaan ajatella olevan näin olevan pieni rakennuspalikka (Sairing 2019). Näin pystytään keskittymään aina oleelliseen ja suurten kokonaisuuksien ja eri riippuvuuksien hallitseminen on helpompaa. Tämä myös mahdollistaa erinäisten testausmenetelmien kohdentamisen ja jaottelun isojen kokonaisuuksien kanssa (Coleman 2017).



Kuva 1. Havainnollistava kuva komponenttilähtöisestä kehityksestä ObSAS 2.0 -projektissa

Oleellista on siis, että luodut komponentit ovat heikosti kytkettyjä (loosely-coupled) itsenäisiä komponentteja ja jokaisella komponentilla itsessään on oma rajapinta (interface), jolla se kommunikoi muun kokonaisuuden kanssa.

Näin kun useat komponentit toimivat, pystytään muodostamaan modulaarisesti toimiva sovellus eli korkean koheesion ja heikon kytkennän mukainen. (Sairing 2019.) Koohesio -ja kytkentä -ajattelu kulkevat siis koko ajan mukana.

Jotta komponenttilähtöinen kehitys toimisi tehokkaasti, on tärkeää tehdä tiettyjä asioita selväksi projektin ja koko kehitysympäristön kanssa. Voidaan puhua koodauskäytännöistä/koodausohjeistuksesta (coding convention) sekä aivan yleisesti säännöistä, miten projektia rakennetaan ja miten tietyt asiat tehdään. Yhtenäistetään käytännöt, niin että kaikki seuraavat niitä. Tämä myös edistää komponenttikirjaston luomista. (Sairing 2019.)

Apuna komponenttilähtöisessä kehityksessä voidaan käyttää myös erilaisia komponenttitutkijoita eli *component explorer*. Nämä ovat eräänlaisia sovelluksia, jotka toimivat projektin sisällä ja niiden avulla voidaan eristetyssä ympäristössä kehittää komponentteja varsinainen projektin tai yleisesti komponenttikirjastoa varten sekä kuvata niitä niiden eri tiloissa ja paljon muuta. Näitä ovat muun muassa Storybook, Styleguidist ja Chromatic. (Nguyen 2017.) Näistä ensimmäiseksi mainittua Storybook-työkalua tulen käsittelemään myöhemmin lisää teoriaosiossa omana lukunaan sekä erityisesti käytännönosiossa esimerkkien avulla.

2.2 Vahvuudet ja heikkoudet

Komponenttilähtöisellä kehityksellä on monia vahvuuksia, mutta myös osaltaan heikkouksia, jotka ovat enemmänkin käyttäjälähtöisiä. Vahvuudet kuitenkin syrjäyttävät heikkouksien tuoman haitan. Selvennetään seuraavaksi mitä nämä ovat.

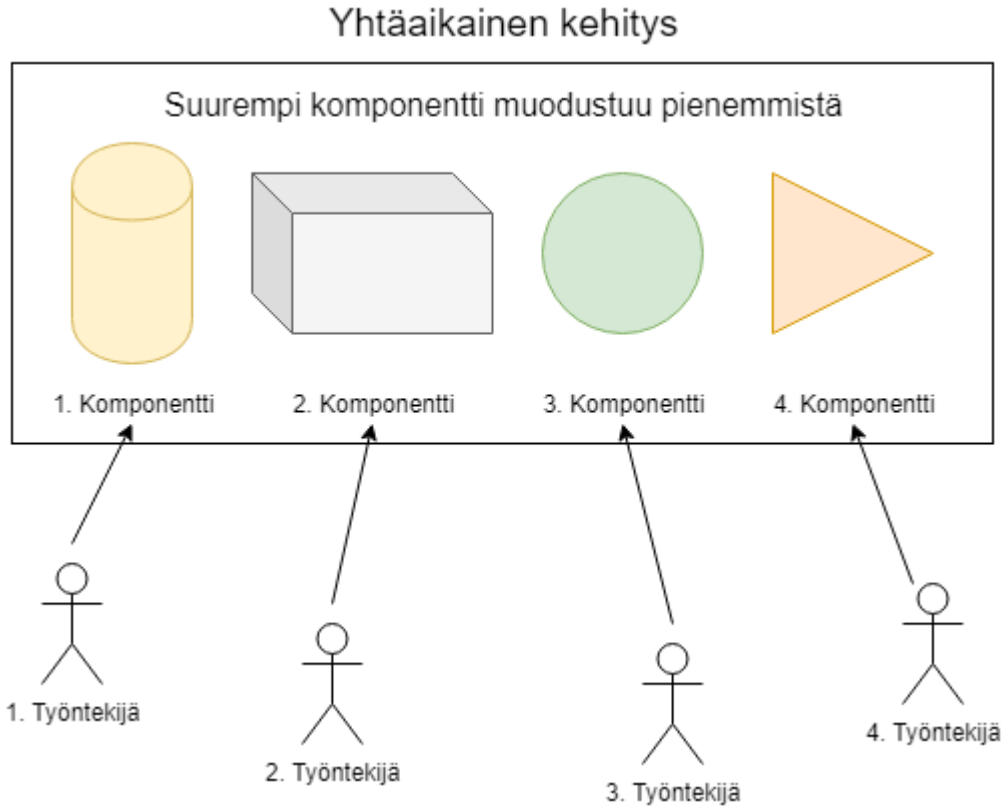
Selkeästi komponenttilähtöisen kehityksen vahvuudet tulevat esiin nykyaikaisten käyttöliittymäkirjastojen tai kehikkojen (frameworks) kanssa. Näitä ovat esimerkiksi jo edellisessä luvussa mainittu React sekä Angular ja Vue. Varsinkin nämä edellä mainitut ovat suunniteltu ja rakennettu juurikin tämänlaisen ajatusmaailman pohjalle (Sairing 2019).

Komponenttilähtöisen kehityksen vahvuuksia on useita. Näitä on uudelleen käytettävyys, parempi ylläpidettävyys, testauksen helpottuminen, kohdennetun palautteen antaminen, nopeampi ja yhtäaikainen kehitystyö sekä mahdollisuus rakentaa komponenttikirjasto. Jotta ymmärrämme näiden perusasiat mitä niillä tarkoitetaan, on niitä hiukan selvennettävä.

Uudelleenkäytettävyys mahdollistaa komponenttien käytön muuallakin kuin vain yhdessä paikassa ilman, että komponenttia tarvitse muokata juuri toiseen tilanteeseen sopivaksi. Parempi ylläpidettävyys taas antaa mahdollisuuden korjausten tai lisäysten kohdentamiseen aina tiettyyn komponenttiin ilman tarvetta lähteä muokkaamaan isompia kokonaisuuksia, mistä voi aiheutua virheitä ja ongelmia muualle. (Sairing 2019.) Lisäksi kohdennettudokumentointi auttaa selkeyttämään rakennetta. Testauksen helpottuminen, eli testit voidaan kohdentaa aina tiettyyn komponenttiin ja testata juurikin näin halutut toiminnallisuudet (Sairing 2019). Näin pystytään minimoimaan jäljittelytiedon (mock-data) antaminen juurikin tarvittuun määrään. Kohdennetun palautteen antaminen mahdollistaa tutkia ja arvioida nimenomaan yhtä tiettyä komponenttia paremmalla tarkkuudella, kun suunnitellaan/arvioidaan komponenttia (Coleman 2017).

Nopeampi ja yhtäaikainen kehitystyö tarkoittaa jaoteltujen isompien kokonaisuuksien nopeampaa kehitystä, kun ne on pilkottu pienempiin osiin ja niitä voidaan kehittää tarvittaessa yhtäaikaisesti isomman työryhmän toimesta (kuva

2), koska niillä on selkeät toiminnallisuudet ja käyttöliittymäsuunnitelmat kuvattuna ja suunniteltuna (Coleman 2017).



Kuva 2. Yhtäaikainen kehitys

Yhtenä tärkeimmistä hyödyistä on tietysti mahdollisuus komponenttikirjaston luomiseen, joka mahdollistaa luotujen komponenttien tarkastelun ja tutkimisen ilman, että ne ovat liitettynä suoraan käyttöön. Myös jakaminen helpottuu kehitysympäristössä. (Sairing 2019.)

Vaikka komponenttilähtöisessä kehityksessä onkin joitakin heikkouksia, ne voivat olla kuitenkin enemmän seurausta siitä, kuinka komponenttilähtöistä kehitystä hyödynnetään/käytetään ja mitä sen voidaan ajatella olevan, eikä niinkään seurausta siitä itsestään. Useampiin näistä ratkaisu löytyykin ajattelu- tai toimintatavan muutoksella. Front-end -kehittäjä Nikolay Kozhukharenko (2017) tuo esiin kaksi selkeää eri ajatusta, jotka hän näkee miksi komponenttilähtöinen kehitys ei useinkaan toimi juuri edellä mainittuihin asioihin viitaten.

Yhtenä heikkoutena voidaan nähdä se, kuinka komponenttikäyttöinen kehitys otetaan käyttöön työskentelyssä (Kozhukharenko 2017). Usein se tapahtuu jo keskeneräisen projektin aikana, jolloin tietysti ongelmia tuovat puolestaan se, miten jo olemassa olevat tuotokset asettuvat tähän ajatusmaailmaan. Paljonko

vanhoja komponentteja joudutaan refaktoroimaan? Miten paljon aikaa kuluu käyttöönottoon ja ylipäätään toiminnan oppimiseen käytännössä? Se ei myöskään itsessään ole ratkaisu ongelmiin tai paranna tuloksia, vaan antaa ajatusmaailman, jota oikein toteutettuna tuottaa oikeita tuloksia.

Toinen mikä ongelma voi tulla esiin, on kuinka komponenttilähtöinen ajatusmaailma oikeasti, tulisi toimia. Konkreettinen käyttö. Usein se ehkä nähdään vain kehitystiimin työkaluna, eikä niinkään liiketoimintaa kehittävänä tapana (Kozhukharenko 2017). Myös se, että sitä käytetään vain apuprojektina eikä työstetä oikeana projektina, syö sen hyötyjä käytännössä ja on näin ollen osa ongelmaa.

3 REACT JA KOMPONENTIT

React on Facebookin työntekijän Jordan Walken kehittämä ja nykyisin avoimeen lähdekoodiin perustuva JavaScript -kirjasto yhden sivun (single page) verkkosovellusten ja myös mobiilisovellusten (React native) tuottamiseen. Se on vuodesta 2013 avoimeen lähdekoodiin perustuva ja sitä ylläpitää Facebook sekä eri yksittäiset kehittäjät, yhteisöt ja yritykset (Hermel 2013). Se on yleisesti hyvin suosittu JavaScript -kirjasto ja sillä on maailmanlaajuisesti paljon käyttäjiä ja kehittäjiä. Suurista yrityksistä Facebookin lisäksi Reactia käyttää muun muassa Instagram, Netflix ja Dropbox (Stackshare 2019).

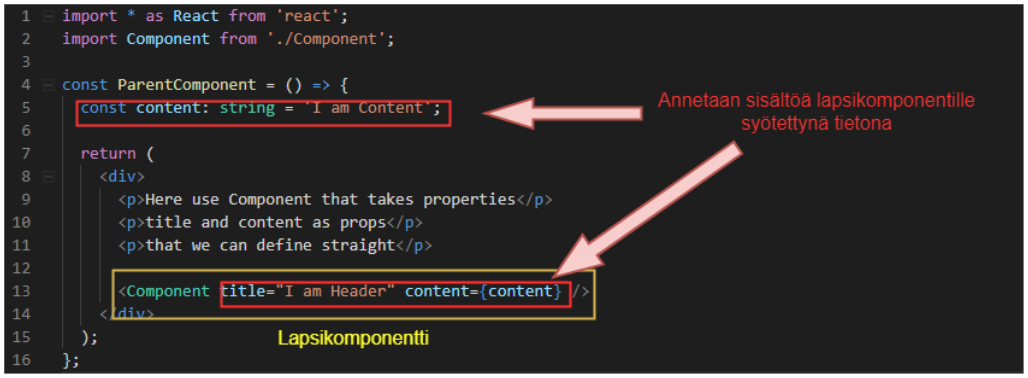
Peruskäytöltään React soveltuu käyttöliittymien luontiin ja nopeisiin tiedon ja tilojen muuttamiseen ja tallentamiseen. Ohjelmointikielenä toimii JavaScript ja myös sen tyypitetty versio TypeScript, jota tämä opinnäytetyö hyödyntää esimerkeissä (React 2019a). Sen perusajatus on komponenteissa. Itsessään React sisältää perustoiminnallisuuden sovellusten luomiseen, mutta yleensä suurempien ja monipuolisempien kokonaisuuksien luomiseen tarvitaan muita ulkoisia kirjastoja.

Seuraavat luvut käsittelevät muutamia Reactin perusasioita, jotka on hyvä ymmärtää ennen varsinaista käytäntöä. Niitä ovat komponentin tila (state) sekä sovelluksen tila (application state = Redux) ja komponentin ominaisuudet (props), siihen miten tilaa käytetään, hallitaan ja päivitetään sekä varsinaiset

komponentit. Näihin asioihin tulen viittaamaan enemmän, kun varsinaiseen komponenttien käsittelyn aika tulee.

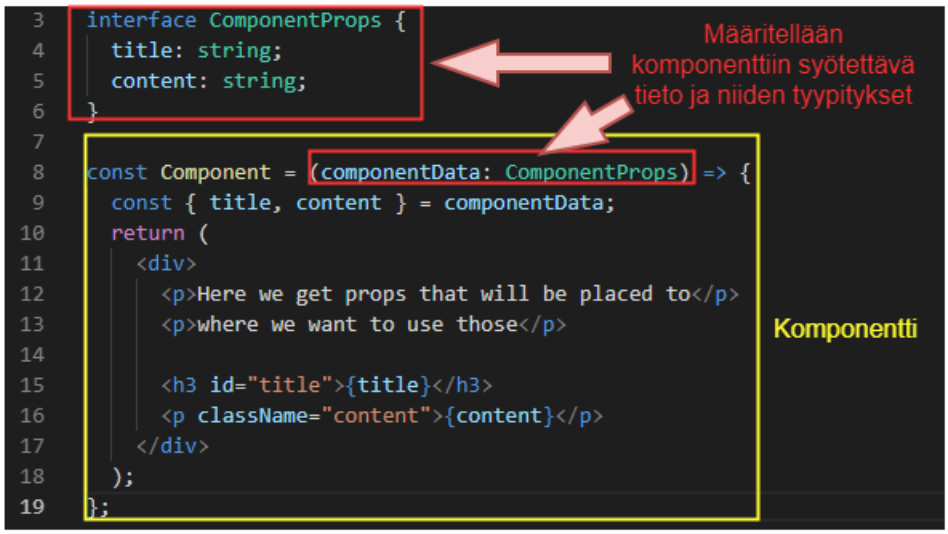
3.1 Komponentin ominaisuudet (props)

Komponentin ominaisuuksilla (props) tarkoitettaan komponentille määriteltyä ja sen ylemmän tason komponentilta (kuva 3) saamaa muuttumatonta tietoa, parametreja, jolla voidaan määrittää alemman komponentin toiminnallisuuksia, antaa käsiteltävää/esitettävää tietoa, joita sitten varsinainen komponentti käyttää.



Kuva 3. Esimerkki tiedon syöttämisestä lapsikomponentille

Kuten voimme alempana olevasta kuvasta (kuva 4) huomata kuinka funktiokomponentti määrittelee ja käsittelee komponentin ominaisuuksia (props) käyttöliittymän (interface) ja tyypitysten avulla. Jotta toiminnallisuudet säästyisivät mahdollisilta virheiltä (bugs), on hyvä määritellä syötettävien tietojen tyypitykset siihen minkä tyyppistä tai minkä muotoista tietoa se voi vastaanottaa.



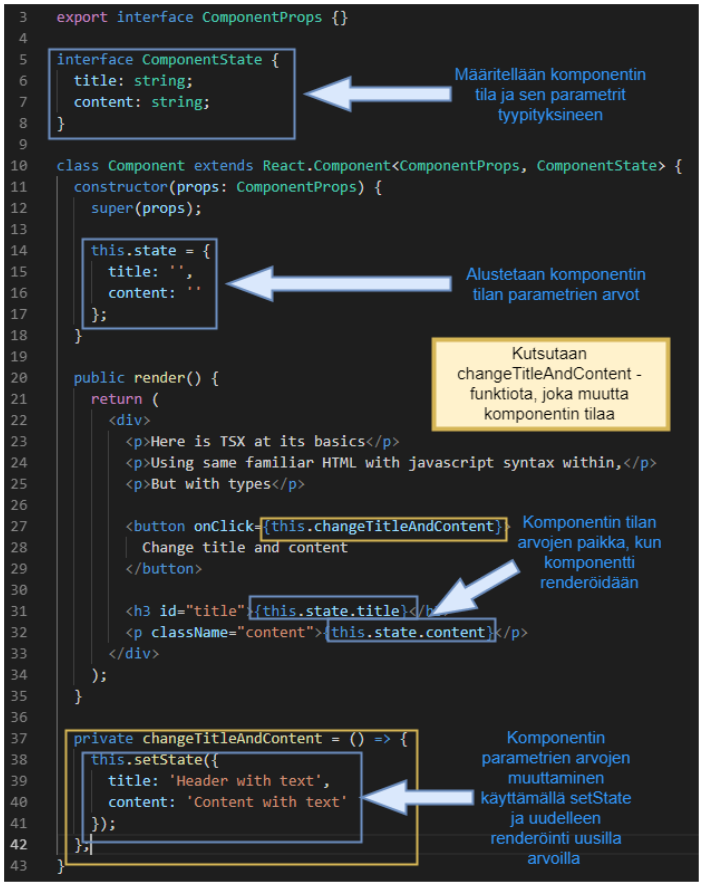
Kuva 4. Esimerkki syötetyn tiedon (props) toiminnasta lapsikomponentissa

Ylempänä oleva esimerkki näyttää kuinka asia tapahtuu TypeScriptiä käyttäen, mutta myös JavaScriptissä voidaan määritellä tietotyypit (PropTypes), jokaiselle tiedon osalle (ks. liite 1) (React 2019b).

Syötettävää tietoa voidaan syöttää myös tarvittaessa usean komponentin läpi syvemmälle komponenttien hierarkiassa. Syötetty tieto voi olla siis muuttujia, objekteja, taulukoita, jopa React -komponentteja yms. sekä metodeja, joiden toiminnallisuus on ylemmän komponentin sisällä, mutta joita kutsutaan nimenomaan lapsikomponentissa (React, 2019c).

3.2 Komponentin tila ja Sovelluksen tila

Komponentin tila eli *state* (kuva 5) tarkoitetaan komponentin sisällä olevaa muuttuvaa/päivittyvää käyttäjän määrittelemään objektimuotoista tietoa, joka halitsee komponentin toiminnallisuuksia ja sisältöä sen elämän eri jaksoissa. Komponentin tilat ovat käytössä ainoastaan luokkakomponenttien sisällä, mutta niiden avulla voidaan hallita tilattomiin komponenttien ominaisuuksien sisältöä (props). (React 2019d.)



Kuva 5. Esimerkki tilan (state) toiminnasta luokka -komponentissa

Komponentin tilaa hallitsemalla komponentin eri elämän jaksoissa pystytään luomaan erittäin monipuolisia toiminnallisuuksia siihen, miten ja mitä komponentissa tapahtuu tietyllä hetkellä. Käytännössä tilan päivitys tapahtuu alkaen *constructor()* -metodissa, jossa komponentin tila alustetaan ja varsinainen päivitys kutsumalla *setState()* -metodia, joka saa parametrina objektimuotoista tietoa. Aina kun tilaa päivitetään *setState()* -metodilla komponentti uudelleen renderöi (re-render) itsensä uusilla tilan tiedoilla, jolloin päivitetyt tiedot tulevat käyttöön (GeeksforGeeks 2019b).

Yleisesti ottaen, kun puhutaan tilasta, sillä tarkoitetaan siis komponentin omaa tila eli aikaisemmin mainittua *state* ja sen hallitsemista tarpeen mukaan. Reactiin sekä Angulariin ja useisiin muihin JavaScript kirjastoihin on esimerkiksi saatavissa myös avoimeen lähdekoodiin perustuva *Redux.js* JavaScript kirjasto, joka mahdollistaa sovelluksen eri komponenttien tilojen hallitsemisen yhdellä koko sovelluksen tilan avulla (Redux 2019). Sen ovat luoneet Dan Abramov ja Andrew Clark vuonna 2015 (Three Devs and a Maybe 2015).

Perusperiaate Reduxissa on tarjota helppo tapa päästä käyttäjän määrittelemään/päivittämään tietoon käsiksi koko sovelluksen laajuisesti kaikissa komponenteissa, jotka ovat linkitettynä siihen. Se myös tarjoaa helpon tavan hallita toimintoja eli *actions*. Ne ovat objektimuotoista tietoa ja kuvaavat mitä Reduxin halutaan tekevän, jonka sitten *reducer* -funktio toteuttaa. Yksinkertaistettuna Reducer -funktiot ottavat siis vastaan vanhan state objektin sekä action ja palauttavat uuden staten (kuva 6). (Sharma 2019.)

Action tyypit enum -muuttujassa

```
export const enum CustomerActionType {
  CUSTOMERS_NEW = 'CUSTOMERS_NEW',
  CUSTOMERS_CANCEL = 'CUSTOMERS_CANCEL',
  CUSTOMERS_UPDATE = 'CUSTOMERS_UPDATE',
}
```

Action -tyypitys

```
export interface UpdateCustomerAction extends AnyAction {
  type: CustomerActionType.CUSTOMERS_UPDATE;
  payload: {
    companyId: CompanyId;
    customer: Customer;
  };
}
```

Action -apufunktio

```
export const updateCustomer: ActionCreator<UpdateCustomerAction> = (
  companyId: CompanyId,
  customer: Customer,
) => {
  return {
    type: CustomerActionType.CUSTOMERS_UPDATE,
    payload: {
      companyId,
      customer,
    },
  };
};
```

Reducer -funktio

```
export interface CustomerState {
  editing: boolean;
  companyId?: CompanyId;

  editedCustomer?: Customer | undefined;
}

const initialState: CustomerState = {
  editing: false,
  editedCustomer: undefined,
};

export const customerReducer: Reducer<CustomerState> = (
  state = initialState,
  action: CustomerAction,
) => {
  switch (action.type) {
    case CustomerActionType.CUSTOMERS_UPDATE: {
      return {
        ...state,
        editing: true,
        companyId: action.payload.companyId,
        editedCustomer: {
          ...action.payload.customer,
        },
      };
    }

    default:
      return state;
  }
};
```

Kuva 6. Esimerkki reduxista

Käytännössä yksi komponentti lähettää kutsun *dispatch* -funktiota käyttämällä (kuva 7), jossa määritellään toiminnon tyyppi sekä tieto mitä toiminto tarvitsee. *Reducer* -funktio lopulta päivittää redux staten uusilla arvoilla. On tärkeää muistaa, ettei vanhaa state objektia saa mutatoida, vaan aina tulee palauttaa uusi state -objekti (Sharma 2019).

```
266   private rowOnClickHandler = (customer: Customer) => () => {
267     const { companyId, dispatch } = this.props;
268     dispatch && dispatch(updateCustomer(companyId, customer));
269   };
```

Kuva 7. Esimerkki dispatch funktiosta

Itse sovellus saa Redux staten käyttöön, kun juuressa oleva index.jsx/tsx -tiedoston sisältämä *App* -komponentti ympäröidään *Provider* -komponentilla, joka antaa propseina redux storen. Näin kaikki tämän alapuolella olevat komponentit näkevät ja pystyvät tarvittaessa käyttämään redux storea. Varsinainen yksittäinen komponentti tarvitsee kuitenkin yhdistää vielä Redux storeen *Connect* -funktion avulla, joka ottaa parametreina *mapStateToProps* funktion palauttaman sisällön. Tämä sisältö pitää sisällään sen sisällön, minkä käyttäjä haluaa sovelluksen tilasta hakea komponentin käyttöön. (Sharma 2019.)

Riippuen, onko sovellus JavaScriptillä tai TypeScriptillä tehty ja onko joitain muita kokonaisuutta yksinkertaistavia keinoja käytetty, Redux toteutukset poikkeavat toisistaan.

3.3 Luokka- ja tilattomat funktiokomponentit

Luokkakomponentit (class) ovat yksi vaihtoehto, kuinka reactissa voidaan luoda erilaisia komponentteja. Käytännössä niitä on kahta eri versioita *createClass* ja *extends* -class komponentteja. Kummatkin eroavat merkintä-, käyttö-, ja -toimintatavoiltaan sekä käyttötarkoituksiltaan toisiinsa nähden enemmän tai vähemmän. Tämä opinnäytetyö tulee kuitenkin käsittelemään jälkimmäistä *extends* (julkaistu React 0.13) -muotoa, koska se on uudempi sekä sen vuoksi, että *createClass* on vanhentunut (React 15.5.0) ja poistunut käytöstä nykyisessä React versiossa (React 2017).

Jo aikaisemmin olemme esimerkkikuvista saaneet huomata miltä *extends* -class -komponentti voi näyttää (Kuva 3.) merkintätapojen yhteydessä ja mitä

toiminnallisuuksia se voi sisältää kuten komponentin tilan (state). Sen käyttö tuli mahdolliseksi ES2015 (ECMAScript 2015) yhteydessä. Yksinkertaisuudessaan se voi olla kuten alempana olevasta kuvasta (kuva 8) voimme nähdä. Hyvinkin yksinkertaista JSX koodia. Syventymällä enemmän, huomaamme sillä olevan muitakin ominaisuuksia, jotka tekevät siitä erittäin hyödyllisen.

```
3  class BasicClassComponent extends React.Component {
4    render() {
5      return (
6        <div>
7          <p>Hi!</p>
8          <p>I'm extended class component at its basic form</p>
9        </div>
10     );
11   }
12 }
```

kuva 8. Esimerkki yksinkertaisesta extends -class komponentista.

Komponentin tilan, jota aikaisemmin käsittelimme lisäksi *extends* -class komponentit hyödyntää erilaisia ”elinkaari” (lifecycle) -metodeja (ks. liite 2). Niiden avulla voimme komponentin elämänkaaren eri jaksoissa tehdä haluttuja asioita. Esimerkiksi kun komponentti latautuu/latautui ensimmäisen kerran tai jopa silloin kun komponentti tuhotaan. Kaikki tämä mahdollistaa asioiden jaksotuksen ja resurssit jakautuvat tasaisemmin ja hallitummin komponentin elinkaaren aikana. (React 2019e.)

Koska osa lifecycle -metodeista on vanhentuneita, eikä niiden käyttöä suositella, on uusimassa Reactin versiossa uusia metodeja, joilla on korvattu nämä tai jaettu niiden toiminnallisuuksia uudelleen järkevämmin ja tietoturvallisesti paremmin. Esimerkiksi *componentWillReceiveProps()* sijaan suositellaan käytettävän *getDerivedStateFromProps()* tai *componentDidUpdate()*. Syyksi tähän on se, etteivät käyttäjät ole aina ottaneet huomiota, milloin ja missä sekä mitä mikäkin lifecycle -metodi saisi sisältää, jotta komponentti ja sovellus rakenteellisesti järkevä ja turvallinen. (React 2019f.)

Tilattomat funktiokomponentit (Stateless Functional) ovat nimensä mukaisesti funktioita, jotka ottavat vastaan parametreina niille määriteltyjä komponentin ominaisuuksia (props) ja palauttavat lopulta React Elementin eli JSX. Yksinkertaisuudessaan niillä pystytään esittämään muotoiltua tietoa halutalla tavalla ja halutuilla tyyleillä (kuva 9), ja koska kyseessä on nimensä mukaisesti tilaton

komponentti, sillä ei ole käytössä komponentin tilaa (state) tai elinkaarimetodeja (lifecycle methods). Syy tähän on, ettei funktiokomponentin sisällä *this* -osoitin kuvaa itse komponenttia (Bertoli 2017, 60).

```
3   interface ComponentProps {
4     title: string;
5     content: string;
6   }
7
8   const Component = (props: ComponentProps) => {
9     return (
10      <div>
11        <h1>{props.title}</h1>
12        <p>{props.content}</p>
13        <p> I'm stateless function component at its basic form </p>
14      </div>
15    );
16  };
```

Kuva 9. Esimerkki yksinkertaisesta funktio komponentista.

Tilattomien funktiokomponenttien vahvuus onkin niiden helpossa käytettävyydessä ja ylläpidettävyydessä sekä testattavuudessa. Ne eivät tee mitään ylimääräistä mitä niiden ei haluta tekevän. Näin ollen ne ovat erittäin käytännöllisiä, kun halutaan luoda uudelleen käytettäviä komponentteja.

Reactin uusimmissa versiossa (React versio suurempi kuin 16.8) on kuitenkin mahdollista käyttää komponentin tilaa ja muita *extends* komponenttien ominaisuuksia koukkujen (hooks) avulla. Tästä syystä ei voida enää puhua suoraan tilattomista funktiokomponenteista, vaan mielellään **funktiokomponenteista**. Esimerkiksi komponentin tilaa voimme hallita *useState* -koukkua käyttämällä kuten alhaalla näemme (kuva 10) (React 2019g).

```
3   interface ComponentProps {
4     title: string;
5   }
6
7   const Component = (props: ComponentProps) => {
8     const [randomNumber, setRandomNumber] = React.useState(0);
9
10    return (
11      <div>
12        <h1>{props.title}</h1>
13        <p>Number: {randomNumber} </p>
14        <button onClick={() => setRandomNumber(Math.random() * 10)}>
15          Random number between 0-9
16        </button>
17      </div>
18    );
19  };
```

Kuva 10. Esimerkin tilan hallinnasta funktio komponentissa *useState* käyttämällä.

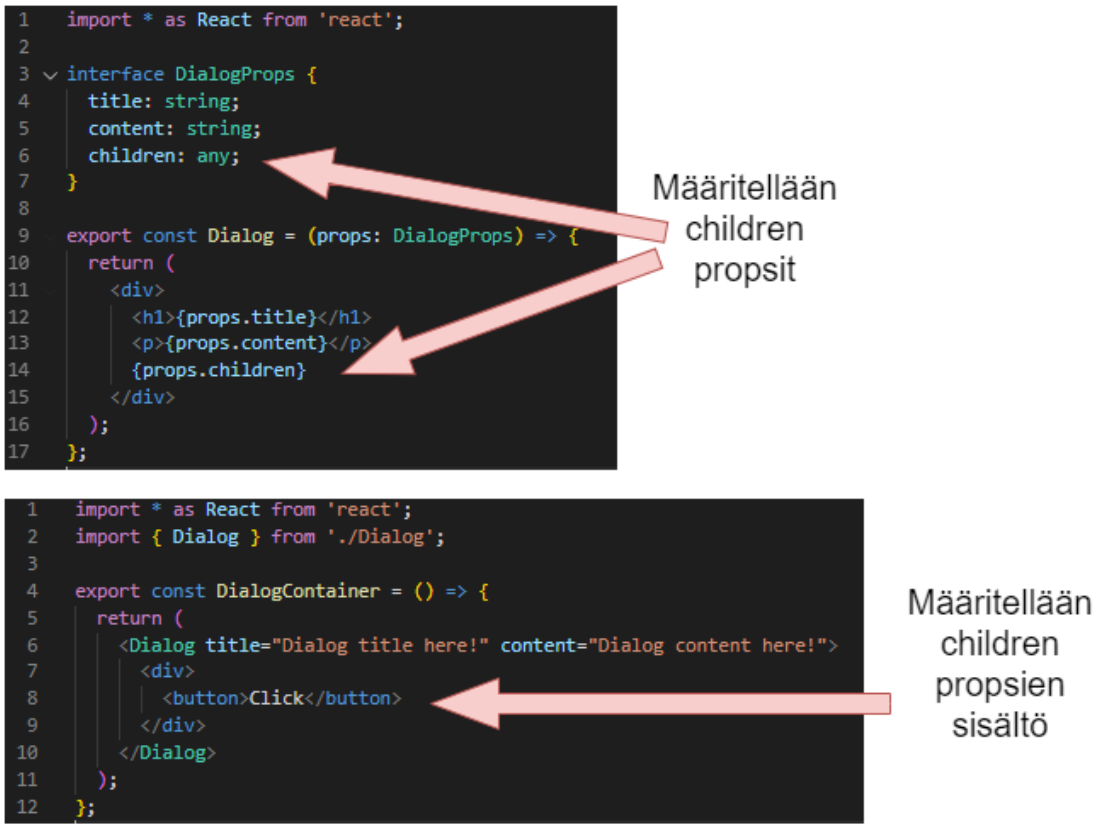
Lisäksi muita koukkuja on efektikoukku (effectHook), jolla pystytään tekemään elinkaarifunktiolle tyypillisiä asioita. Esimerkiksi päivittämään asioita, kun komponentti on asetettu tai komponentti on päivitetty. On myös mahdollista suunnitella omia koukkuja jakamaan toimintoja uudelleen, niin että niitä voi käyttää muissakin komponenteissa. Jotta myös koukkujen käyttö olisi järkevää niihin on myös omat säännöt, joista kerrotaan enemmän Reactin omissa dokumentaatioissa.

3.4 Kompositio vs periyttäminen

Yleisesti kompositiolla ja periyttämisellä tarkoitetaan siis tapaa, kuinka komponentit muodostetaan ja rakennetaan. Molemmat kuitenkin pyrkivät koodin uudelleen käytettävyyteen ja selkeämpään rakenteeseen (Ravivhandran 2019).

Periyttämisessä lapsikomponentti perii äitikomponentin ominaisuudet *extends* -ominaisuutta käyttämällä. Tällöin kaikki mitä äitikomponentissa on määritelty, funktiot ja muut ominaisuudet siirtyvät lapsikomponentille käytettäväksi (Ravivhandran 2019). Esimerkki tästä voisi olla komponentit Ajoneuvo sekä Auto*,* joka tarvitsee samoja ominaisuuksia kuin Ajoneuvo -komponentille on määritelty. Tässä tapauksessa Auto komponentti voi periä Ajoneuvo komponentille määritellyt ominaisuudet class Auto *extends* Ajoneuvo määrittelyllä, kun lapsikomponentti muodostetaan. Samalla periaatteella luokka komponentit periytetään aina React.component, jotta saadaan sen ominaisuudet käyttöön.

Kompositiossa eli koostetussa rakenteessa lapsielementit sekä muu syötettävä voidaan antaa komponentille määrityltyjen ominaisuuksien kautta, sekä erityisen *children* ominaisuuden välityksellä (kuva 11). Komponentti muodostetaan siis pienempiä komponentteja käyttämällä. Käytännössä children ominaisuutta käytettäessä tämä tarkoittaa, että kun komponenttia käytetään, se on JSX-koodissa parillinen tagi ja tämän komponentin tagien sisään tuleva JSX-koodi tulostetaan children ominaisuutena komponentin sisällä sinne missä se on määritelty. Tämä toimii molempien komponenttityypin kanssa. (React 2019h.)



Kuva 11. Kompositio komponentti esimerkki

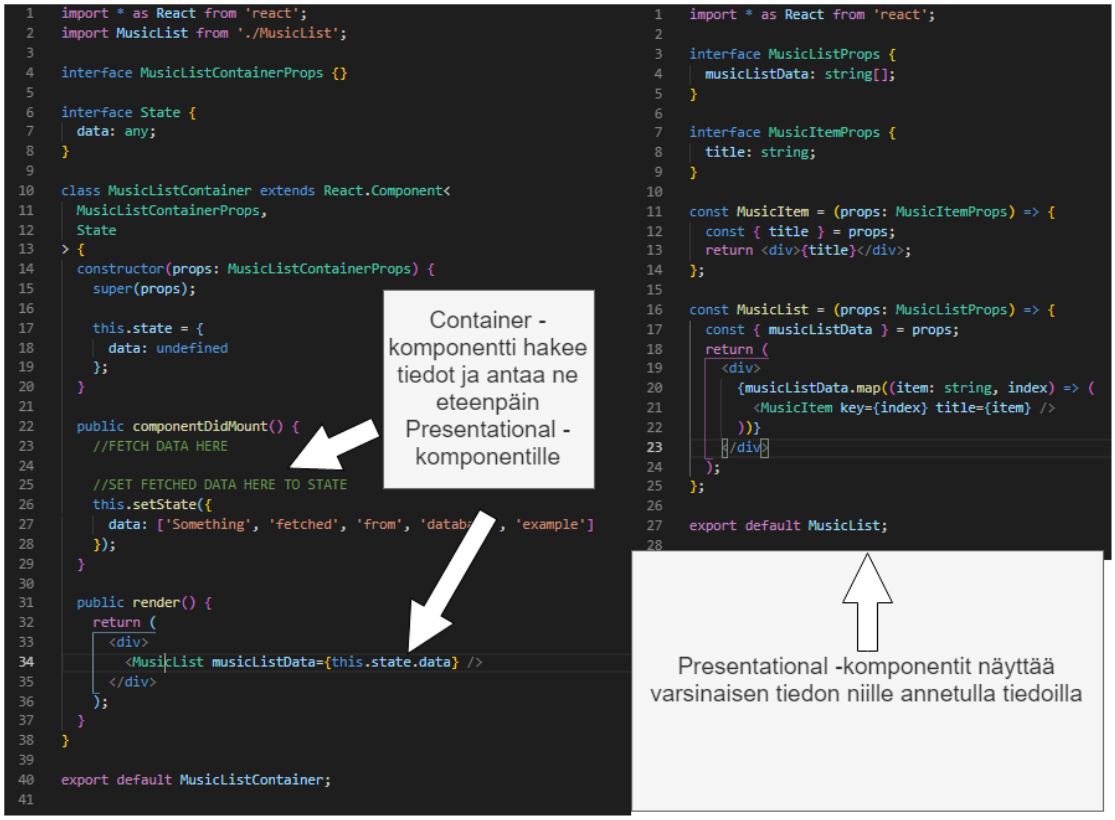
Tämänlainen ratkaisu antaa hyvän liikkumavaran, kun komponentteja suunnitellaan ja mahdollistaa näin ollen uudelleen käytettävien komponenttien luomisen. Yleisesti ottaen se sopii paremmin Reactin komponenttirakenteeseen (Ravivhandran 2019). Myös Reactin omat kehittäjät suosivat omissa dokumentaatioissaan (2019h) kompositio rakennetta, eivätkä näe tarvetta suositella periyttämisrakenteiden käyttöä.

3.5 Komponenttien suunnittelumallit

Kun suunnitellaan ja kehitetään komponentteja Reactissa, on hyvä huomioida mitä ja miten eri komponenttityypit pystyvät hyödyntämään ohjelmointirajapintoja (API, application programming interface). Reactin komponenteilla näitä on yhteensä viisi, jotka ovat komponentin tila (state), syötetty tieto (props), renderointi (render), konteksti (context), ja elinkaaritapahtumat (lifecycle events). Komponenttityypeistä molemmat käyttävät enemmän ja vähemmän toisia näistä edellä mainituista rajapinnoista, riippuen omista ominaisuuksistaan, joita jo aikaisemmin käytiin läpi komponenttityyppi kerrallaan. Tilattomat komponentit käyttävät pääasiassa syötettävää tietoa, kontekstia sekä renderiä,

kun taas tilalliset komponentit pääasiassa komponentin tilaa, elinkaaritapahtumia sekä renderiä. Koska tämä ominaisuuksien jakautuminen vaikuttaa siihen, miten komponenttia voidaan käyttää, se vaikuttaa myös siihen miten isommat kokonaisuudet kannattaa luoda. Tätä varten on olemassa erityisiä suunnittelumalleja, miten tietyt komponentit kannattaa tehdä, jotta ne muodostavat sopivia toiminnallisia kokonaisuuksia niiden ominaisuuksien täydentäessä toisiaan. Näitä ovat kontti (container)-, esittävä (presentational)-, Ylemmän tason (High order) - ja palauttava renderointi (Render callback) komponentti. (Whatley 2018.)

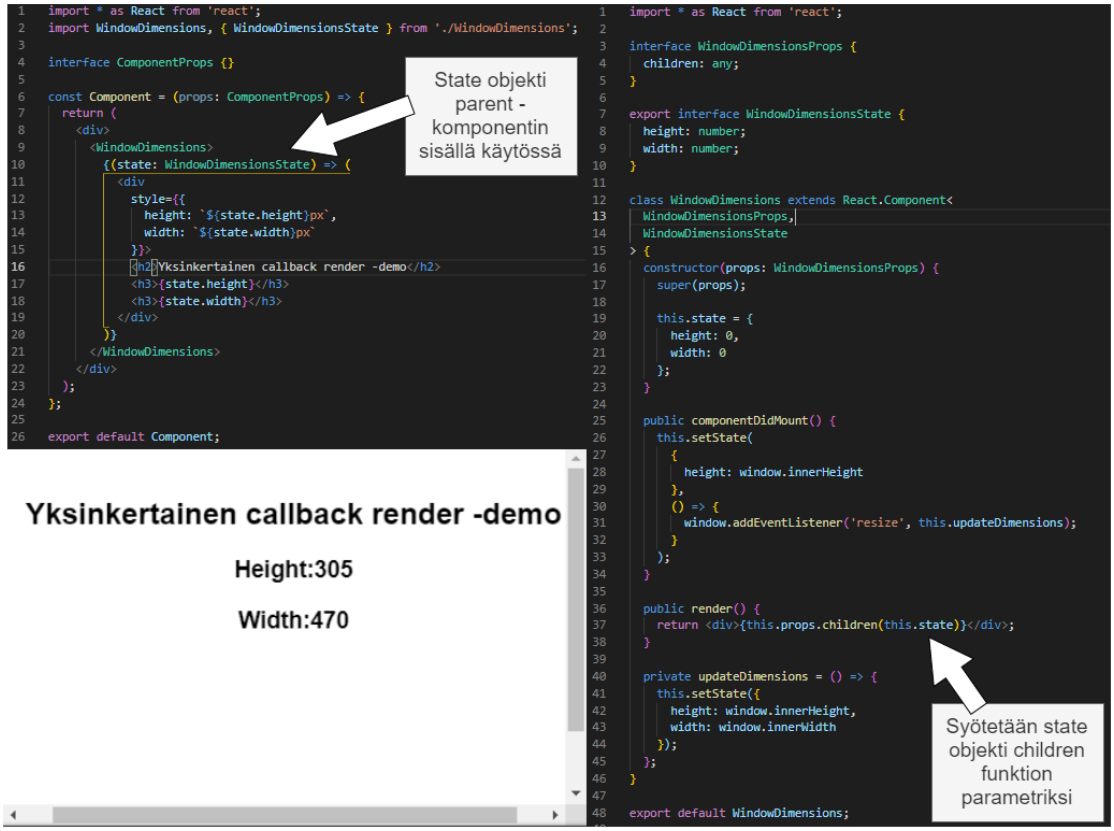
Ensimmäinen näistä mainituista, konttikomponentti eli *Container component* on lähtökohtaisesti luokkakomponentti. Sen toiminnallisuuksiin kuuluu tiedon käsittelyyn ja hakuun sekä lopulta sen jakamiseen liittyviä toimintoja, jotka jaetaan lopulta alemmille, toisena mainituille esittäville komponenteille (Presentantional component), jotka voivat olla funktiokomponentteja tai luokkakomponentteja. Se voidaan myös esimerkiksi liittää osaksi sovelluksen tilanhallintaa kuten Reduxiin. Tämä mahdollistaa sen, että esittävä komponentti, joka ottaa vain vastaan tietoja, voidaan näin erottaa omaksi kokonaisuudeksi ja käyttää uudelleen toisilla tiedoilla. Näin ne yhdessä kapseloivat logiikan ja esittämisen haluttuun tilanteeseen sopivaksi (kuva 12). (Whatley 2018.)



Kuva 12. Esimerkki Container ja Presentational -komponenteista

Ylemmän tason komponenteilla eli *High Order Component* (HOC) voidaan puolestaan esimerkiksi hakea tai tarjota tietoa. Käytännössä Ylemmän tason komponentti on funktio, joka ottaa vastaan komponentin ja palauttaa uuden komponentin. Tämä uusi komponentti omaa ylemmän komponentissa määritetyt ominaisuudet, jotka se on perinyt siltä. (Whatley 2018.) Enemmän tietoa Ylemmän tason komponenteista löytyy Reactin (2019i) omista dokumentaatioista.

Viimeisimpänä mainittu palauttavassa renderöinnissä eli *render callback* määritellään komponentille *children* -ominaisuus funktiona, joka saa parametrina arvon tai arvoja. Kun komponenttia käytetään, sen komponenttitagien sisään määritellään palauttava funktio eli *callback* -funktio *children* ominaisuutena, jolloin saada aikaisemmin komponentin sisällä määritelty arvo käytettäväksi. (Whatley 2018.) Tämän avulla voidaan esimerkiksi tehdä ehdollisia renderöintejä jos jokin arvo täyttää tai ei täytä ehtoja (kuva 13).



Kuva 13. Käytännön esimerkki callback render -mallista

Näiden lisäksi on olemassa tyyli komponentteja (styled component) sekä Proxy -komponentteja. Tyyli komponenteilla voidaan erikoistaa jonkin jo aikaisemmin tehdyn komponentin tyylit vastaamaan haluttua tilannetta. Proxy -

komponentilla voidaan puolestaan ympäröidä jokin toinen komponentti container -komponentin tavoin ja määrittää sille tietyt ominaisuudet, jotka sillä on aina valmiiksi määritelty.

4 STORYBOOK

Storybook on käyttöliittymien ja käyttöliittymäkomponenttien luomiseen, esittelyyn, testaukseen ja dokumentointiin tehty, avoimeen lähdekoodiin perustuva työkalu. Se mahdollistaa komponenttien kehityksen eristetyssä ympäristössä ja antaa näin liikkumavaraa komponenttien luomisessa. Nykyään Storybook on eri kehittäjistä ympäri maailmaa muodostuvan yhteisön kehityksessä.

Sen kehityksen aloitti alkujaan Kadira -niminen startup-yritys Sri Lankasta, jota johti Arunoda Susirapala pienen lahjakkaan kehittäjätiiminsä kanssa ja ensimmäinen versio, Storybook 1.0 julkaistiin vuonna 2016 huhtikuussa. Syyskuussa 2016 julkaistuun Storybook 2.0, joka sisälsi kattavat dokumentaatiot ja hyvän laajennettavuuden sekä maksulliset palvelut, saaden samalla uusia ominaisuuksia ja tukia muille framework-kirjastoille. Kuitenkin saman vuoden joulukuussa Kadira yrityksenä lakkautettiin yllättäen ja Storybook:n kehitys keskeytyi. (Shilman 2017.)

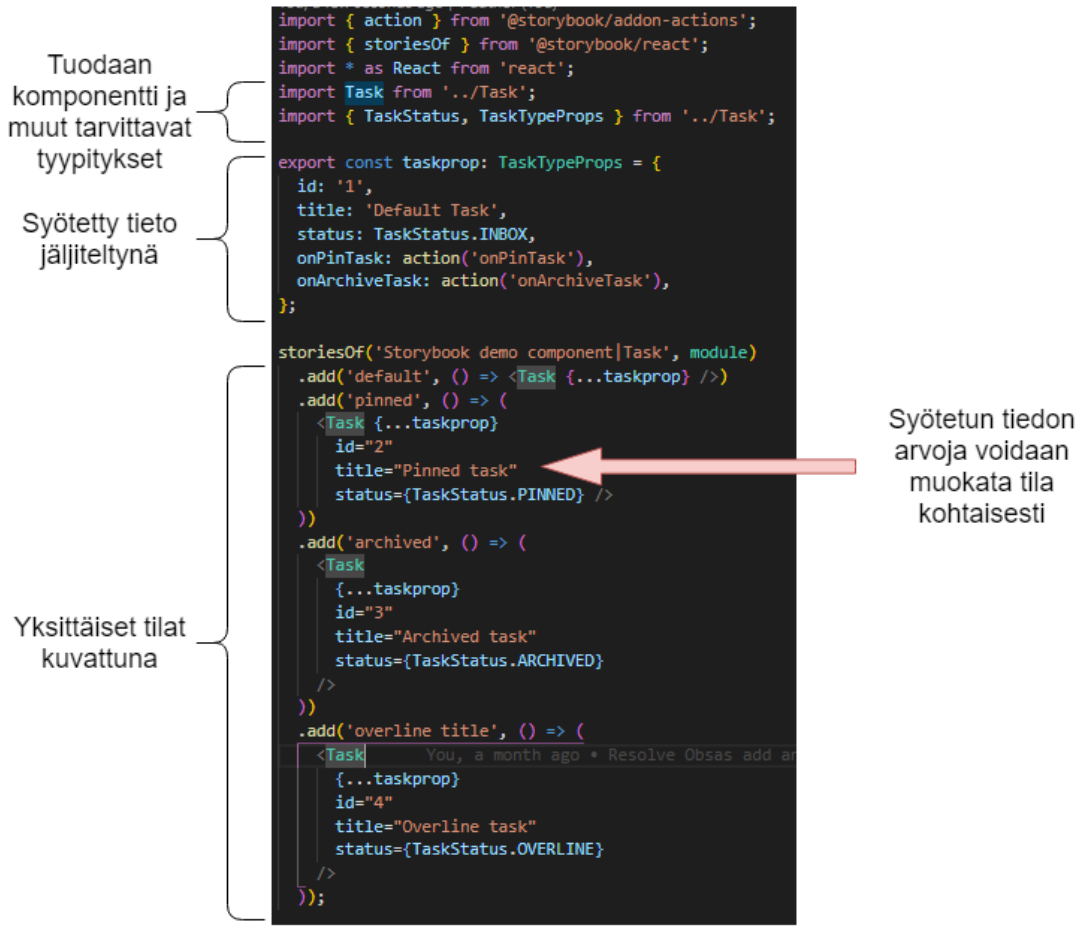
Vuonna 2017 keväällä pienen painostuksen jälkeen, projekti annettiin eri kehittäjistä muodostuvan yhteisön kehitettäväksi. Tähän vaikutti suuresti tunnetun React kehittäjän ja Storybookin tukijan Dan Abramov ehdotus. Storybookin aikaisempi kannattaja Robert de Langen otti ohjat käsiin, kun kehitystä yhteisökehityssuuntaan aloitettiin toteuttaa. Entisestä kehittäjä tiimistä Arunoda Susirapala ja Muhammed Thannis auttoivat tässä siirtymässä ja jatkoivat edelleen projektin ohjauksessa. (Shilman 2017.)

Toukokuussa 2017 julkaistuun Storybook 3.0, joka oli täysin tämän uuden kehitysyhteisön kehittämä. Tämä versio ei ominaisuuksiltaan ollut niinkään suuri, vaan enemmänkin merkkipaalu sille siirtymälle, jota oli ajettu, kun toiminta/kehitys siirtyi Kadiralta yhteisön kehitettäväksi (Shilman 2018). Myöhemmin vuonna 2018 julkaistiin Storybook 4.0 ja nykyinen versio Storybook 2019. Nämä versiot sisälsivät jo enemmän muutoksia ja ominaisuuksia itse työkaluun ja sen mahdollisuuksiin sekä yleisesti työkalun ulkoasuun.

4.1 Perustoiminta

Storybook on käyttöliittymien ja niiden komponenttien luomiseen kehitetty ympäristö. Se mahdollistaa käyttäjän luoda ja kehittää komponentteja interaktiivisesti eristetyssä ympäristössä itsenäisesti ilman varsinaisen sovelluksen liitännäisyyksiä ja vaatimuksia. Samalla se myös mahdollistaa komponenttien toimintojen esittelyn ilman muun sovelluksen tuomaa painolastia (ks. liite 3). (Storybook 2019a.)

Käytännössä perustoiminnallisuus on tarinoiden eli *stories* ympärillä. Halutusta komponentista muodostetaan *component.stories.jsx/tsx* tiedosto, jonka sisällä itse komponentti tuodaan ja siitä ryhdytään muodostamaan tarinoita. Nämä tarinat ovat funktioita, jotka kuvaavat komponentin toimintaa sen eri tiloissa palauttamalla renderöidyn version komponentista halutuilla komponentin ominaisuuksien arvoilla (kuva 14). (Gift 2019.)



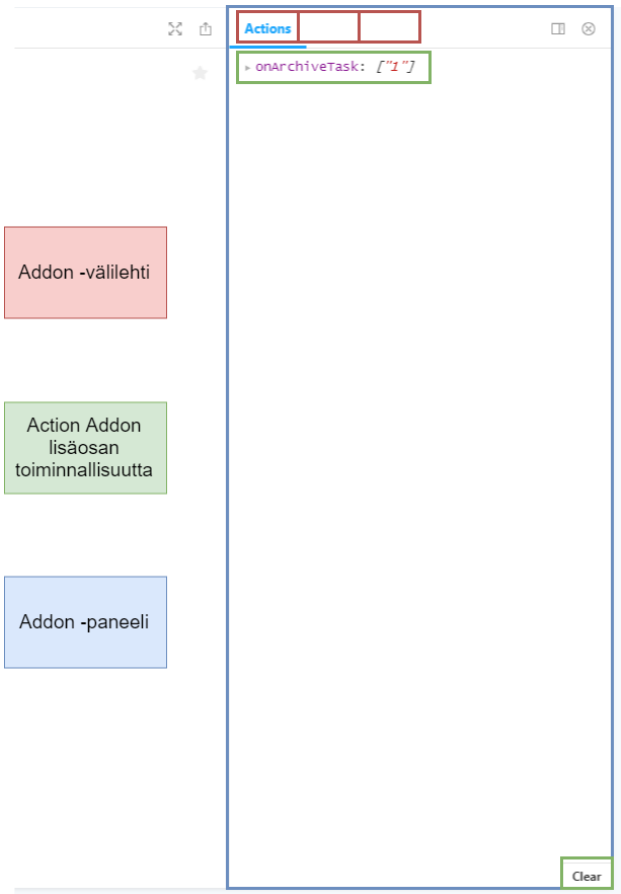
Kuva 14. Esimerkki stories tiedostosta TypeScript esimerkein.

Päämääränä on, että yksi tarina kuvaa komponentin yhtä tilaa ja toinen toista. Tilat voivat olla mitä vain komponentin toiminnallisuuteen tai ulkoasuun liittyviä asioita. Tilanteita on esimerkiksi, mitä tapahtuu jos syötettyä tietoa ei ole tai jos jokin syötetty tiedon osa puuttuu? Tästä tilojen kuvaamisesta voidaan myös käyttää nimitystä Storybook lähtöinen kehitys, kun se viedään tarpeeksi pitkälle.

Storybook lähtöisessä kehityksessä (Storybook driven-development) aikaisemmin mainitut tarinat eli *stories* ovat kuin visuaalisia testitilanteita. Tarkoitus tämän mukaisessa kehityksessä on tehdä tarinat etukäteen ennen varsinaista komponentin työstämistä, eli kuvataan tilat, joita tulevalla komponentilla odotetaan olevan ja toteutetaan ne komponentin työstämisen aikana. (Schwartz 2017.) Periaate toimii samalla tavalla kuin testauslähtöisessä kehityksessä (Test-driven-development).

4.2 Lisäosat ja tilanteet

Koska Storybookin perustoiminnallisuus itsessään on hyvinkin yksinkertainen, voidaan sen ominaisuuksia tarvittaessa laajentaa erilaisilla lisäosilla eli *addons* (kuva 15). Ne voivat olla Storybookin kehittäjien tai ulkopuolisten kehittäjin luomia. Nämä mahdollistavat erilaisten uusien työtapojen ja toimintojen hyödyntämisen. Niitä on mm. Knobs, StoryShots, Actions ja Notes. (Storybook 2019b.) Lisäosia tarkastellaan tarkemmin käytännönosiossa, kun rakennetaan Storybook -työtilaa ja huomioimme mitkä näistä lisäosista voivat tehostaa työskentelyä ja antaa hyödyllisiä ominaisuuksia kehitystä ajatellen.



Kuva 15. Addon -paneeli

Tilanteita, missä Storybookin ominaisuudet tulevat parhaiten esiin ovat muun muassa, kun halutaan esimerkiksi luoda komponenttikirjasto. Storybookin avulla on mahdollisuus järjestellä komponentit niin, että ne ovat saatavissa kaikki yhdestä paikasta ja niiden eri tilojen testaaminen helpottuu. Toinen tilanne, mihin Storybook on hyvä työkalu, on kun halutaan kehittää suunnitella järjestelmä, jossa useat komponentit ovat yhteensopivia ja uudelleen käytettäviä keskenään. Myös visuaalinen testaus ja mahdollisuus jakaa luotu Storybook ja sen sisältö muun suunnittelija- tai kehitysryhmän kesken on tilanteita, joissa Storybookin ominaisuudet mahdollistavat kommenttien ja arvioinnin ryhmän kesken helpommin ja yleisesti yhteistyö ryhmän tai ryhmien kesken helpottuu. (Gift, 2019.)

4.3 Storybook testauksessa

Testauksella ja testilähtöisellä kehityksellä (Test driven-development) on selkeä merkitys, kun kehitetään käyttöliittymäkomponentteja tai mitä tahansa toiminnallisuuksia. Niiden avulla löydetään mahdolliset virheet (bugs), varmistetaan ettei mikään hajoa, kun uudet muutokset tulevat voimaan sekä samalla

se on eräänlainen elävä dokumentaatio toiminnallisuudesta. Storybook ei itsessään ole työkalu testaukseen, mutta siihen saatavat aikaisemmin mainitut lisäosat auttavat tässä ja mahdollistavat toiminnallisuuden laajentamisen testaukseen ja mahdollistaa myös eri tekniikoiden hyödyntämisen siinä. (Storybook 2019c.)

.

Testaus käytännössä tapahtuu useasta eri näkökulmasta katsottuna. Näitä ovat rakenteellinen (structural), toiminnallinen (interaction) ja tyylien (css/style) testaus. Myös perinteisempi käytännön (manual) testaus on selkeä osa näitä näkökulmia ja on tärkeä siinä missä muutkin. Jokainen näistä testauksennäkökulmista tekevät yhtä asiaa ja muodostavat eheän testauskokonaisuuden.

Rakenteellinen testaus on kaikessa yksinkertaisuudessaan komponentin rakenteen testausta. Siinä pyritään testaamaan, että komponentilla on kaikki tarvittavat rakenteelliset ominaisuudet/elementit, jotka sille on määritelty ja luotu. Tätä varten testauksessa on yleisesti käytössä *Snapshot* -testausmenetelmä. Siinä testattavasta rakenteesta otetaan ensimmäisellä testauskerralla kopio rakenteesta ja verrataan tulevaisuuden testeissä aina siihen rakenteeseen. Mikäli muutoksia on, testien tulos ilmoittaa siitä ja jos muutokset ovat tarkoituksellisia voidaan snapshot päivittää vastaamaan uutta rakennetta. Storybookia auttaa tässä StoryShots -lisäosa, jolla tämä pystytään automatisoimaan Snapshot -testaus, jota käsitellään käytännönosiossa enemmän. (Storybook 2019d.)

Toiminnallisessa testauksessa, nimensä mukaisesti testataan toiminnallisuutta. Siinä pyritään simuloimaan komponentin toiminnallisia osia, jotta pystytään näkemään miten ne toimivat ja toimivatko ne oikein. Tätä varten testaukseen on olemassa *Enzyme*-työkalu, joka toimii osana isompaa testausympäristöä, esimerkiksi *Mocha* tai *Jest*. Storybookiin on saatavissa lisäosana *Specifications Addon* -lisäosa, joka mahdollistaa testien kirjoittamisen suoraan tarinoihin. (Storybook, 2019e.) On kuitenkin huomioitavaa, että tällä lisäosalla ei ole pysyvää ylläpitoa tällä hetkellä, joten sen käyttö ei välttämättä ole suotavaa (GitHub 2019a).

Tyylien testauksessa otetaan huomioon mahdolliset muutokset tyyleissä. Näin pystytään varmistumaan siitä, että halutut muutokset ovat tarkoituksellisia eikä

siinä keskitytä, kuinka tyylit on saatu aikaan, joten mahdolliset koodilliset muutoksia tyylien aikaan saamiseksi eivät vaikuta, vain lopputulos. Samalla voidaan puhua myös automatisoidusta visuaalisesta testauksesta. Päämääränä on siis saada käyttäjälle näkyviin mahdolliset muutokset. Haasteena tässä on se, kuinka muutoksia tarkastellaan. Ihmissilmällä muutoksia ei välttämättä näe, mutta pikselitasolla muutoksia saattaa olla paljon. Pikselitason muutokset saattavat olla seurausta eri selaimista ja niiden versioista sekä laitteistosta, joka on vastuussa kuvan esittämisestä käyttäjälle. On myös hyvä pitää mielessä, että tämänlaiset testit ovat huomattavasti raskaampia kuin monet muut. Storybookissa automatisoitua visuaalista testausta pystytään hyödyntämään myös lisäosien kautta, joista osa on maksullisia, kuten *Chromatic* ja osa ilmaisia, esimerkiksi aikaisemmin mainittu *Storyshots* ja sitä laajentava *Storyshots-Puppeteer*, joiden ominaisuudet vaihtelevat toisiinsa nähden paljonkin. (Storybook 2019f.) Käytännön osiossa esitellään jälkimmäisen lisäosan toimintaa enemmän.

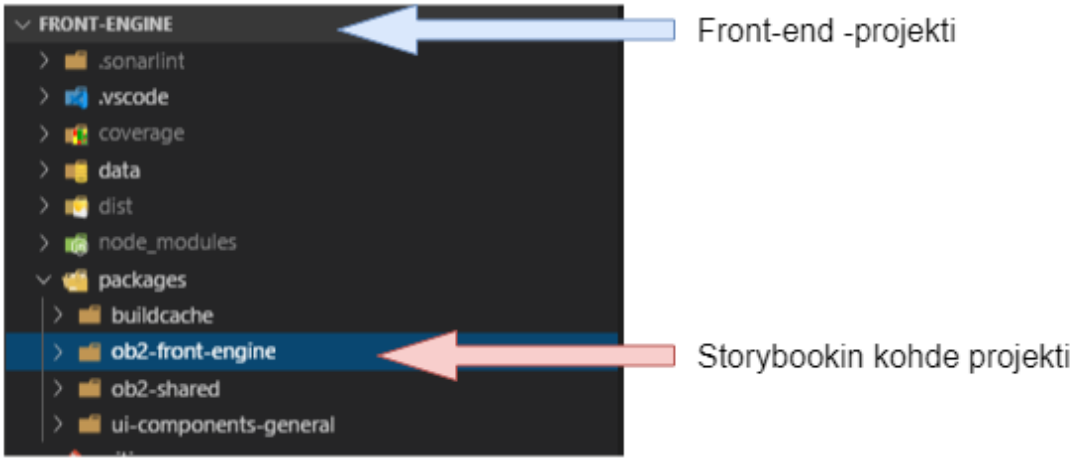
Perinteisin näistä testausmenetelmistä on selkeästi manuaalinen testaus. Storybookilla se voidaan toteuttaa perusominaisuuksien kautta. Tämä tarkoittaa sitä, että pyritään luomaan yksi tai useita kattavia Storybook kokonaisuuksia, joista toisessa komponentit on käyty läpi tarinoissa ja niiden tilat on esitelty. Toiseen puolestaan muodostetaan isompia kokonaisuuksia, joissa komponentit ovat käytössä. Vaihtoehtoisesti yksittäistä komponenttia varten voidaan erillistä Storybook säilytyspaikkaa verrata varsinaisen sovelluksen Storybook sisältöön. Nyt voidaan verrata komponentti kerrallaan kahden eri verrattavan sisällön välillä ja etsiä eroavaisuuksia. (Storybook 2019g.)

5 STORYBOOK OBSAS 2.0 -PROJEKTISSA

Tässä luvussa käsitellään kuinka, Storybook-työkalu asennetaan ja otetaan käyttöön varsinaisessa React -projektissa ja kuinka sitä käytetään komponenttien luomisessa, dokumentoimisessa kuin myös testauksessa. Samalla käydään myös läpi, mitkä eri lisäosat ovat tarpeellisia kehityksen kannalta. Rakennetaan siis täysi komponenttien luomiseen sopiva Storybook -työtila ja esitetään esimerkkikomponentin avulla sen toimintaa. Esimerkkiprojekti ObSAS 2.0 käyttää Reactia ja TypeScriptiä, joten toteutus on myös sen mukainen.

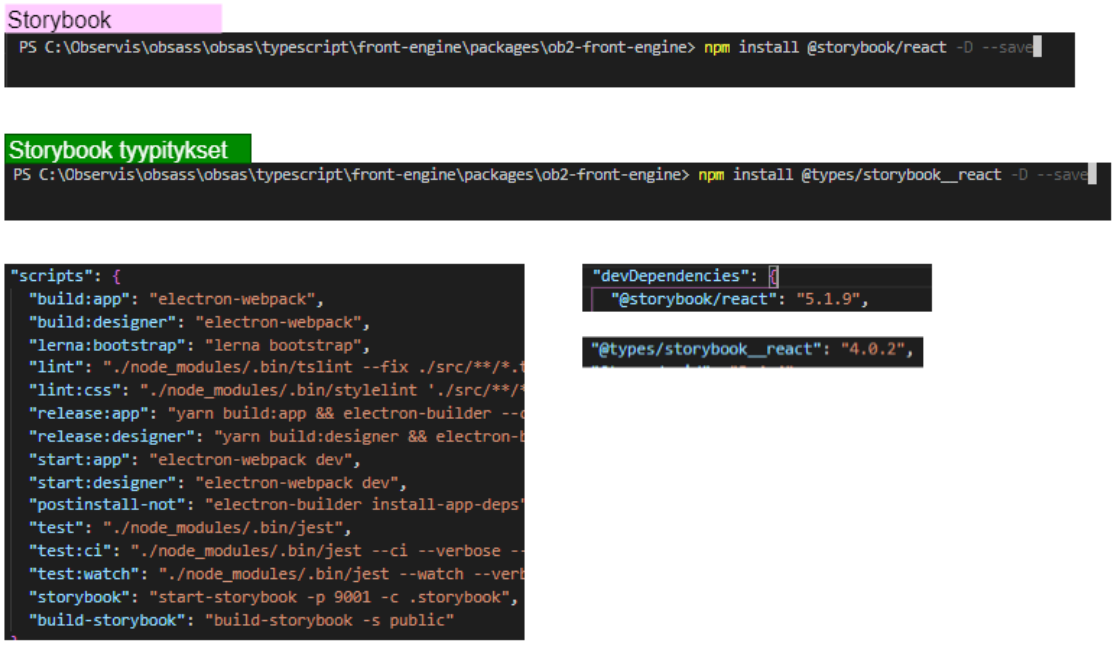
5.1 Storybook työympäristön asennus ja konfigurointi

ObSAS 2.0 kohdalla Storybook asennettaan jo olemassa olevaan projektiin ja koska kyseinen projekti sisältää front-end -ja backend -puolet eriteltynä saman projektin sisällä sekä muutamia muita projektiin kokonaisuuteen vaikuttavia asioita, tulee löytää oikea kohdepaikka Storybook asennukselle. Front-end -projektin alta löytyy vielä erikseen käyttöliittymän luomiseen tarkoitettu projekti, ob2-front-engine, joten Storybook asennettaan sen juureen (kuva 16). Asennuskomennoissa huomioitavia asioita, kun käytetään npm pakettien hallintaa (package manger), kuten lisäkomento *--save*, tarkoittaa että moduuli halutaan tallentaa projektiin ja *-D* tai *-dev* erityisesti kehittäjien liitännäisiin, joita ei huomioida, kun projektista viimein luodaan koontiversio. Erityisesti tyypitykset ja sellaiset liitännäiset, joiden toiminnot eivät vaikuta varsinaisen sovelluksen toimintaan kuuluvat kehittäjien liitännäisiin.



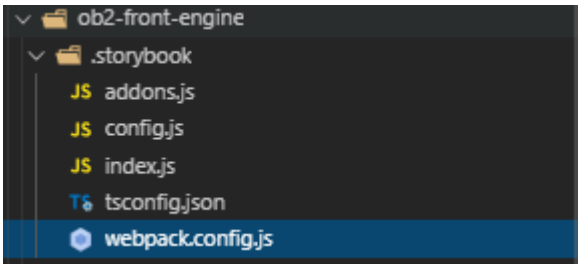
Kuva 16. Esimerkki ObSAS 2.0 kansiorakenteesta.

Varsinainen Storybookin asennusprojekti ja tarvittavat asennuskomennot sille ja sen tyypityksille voidaan nähdä alla olevasta kuvasta (kuva 17). Tämän jälkeen projektin *package.json* kehittäjien liitännäisyyksissä pitäisi olla viittaukset asennettuihin moduuleihin. Package.jsoniin määriteltävissä ajettavissa komennoissa pitäisi olla myös komennot Storybookin käynnistämiselle ja sen koontiversion luomiselle, joskus ne eivät tule automaattisesti, vaan ne on lisättävä käsin.



Kuva 17. Esimerkki asennus sisällöstä

Nyt kun Storybook on saatu asennettua, tulee se konfiguroida toimimaan halutulla tavalla. Ob2-front-engine -projektin sisälle luodaan aivan sen juureen *.storybook* -kansio, jonka sisään tulee määrittelytiedostot TypeScriptille (tsconfig.json), Storybookin lisäosille (addons.js), Storybookille itselleen (config.js). Projektikohtaisesti, kuten tässä tapauksessa tarvitaan myös erityisestä moduulien paketoijaa (moduler bundler) ja sen määrittelytiedostoa (webpack.config.js) sekä muita siihen liittyviä muita tiedostoja, joihin voidaan määritellä jäljittelyfunktioita esimerkiksi kehitysympäristökohtaisia tilanteita, kuten Electronia varten. Lähes kaikki määrittelytiedostot ovat tässä tilanteessa JavaScript tiedostoja TypeScriptin määrittelyä lukuun ottamatta (kuva 18).



Kuva 18. Esimerkki kansiorakenteesta

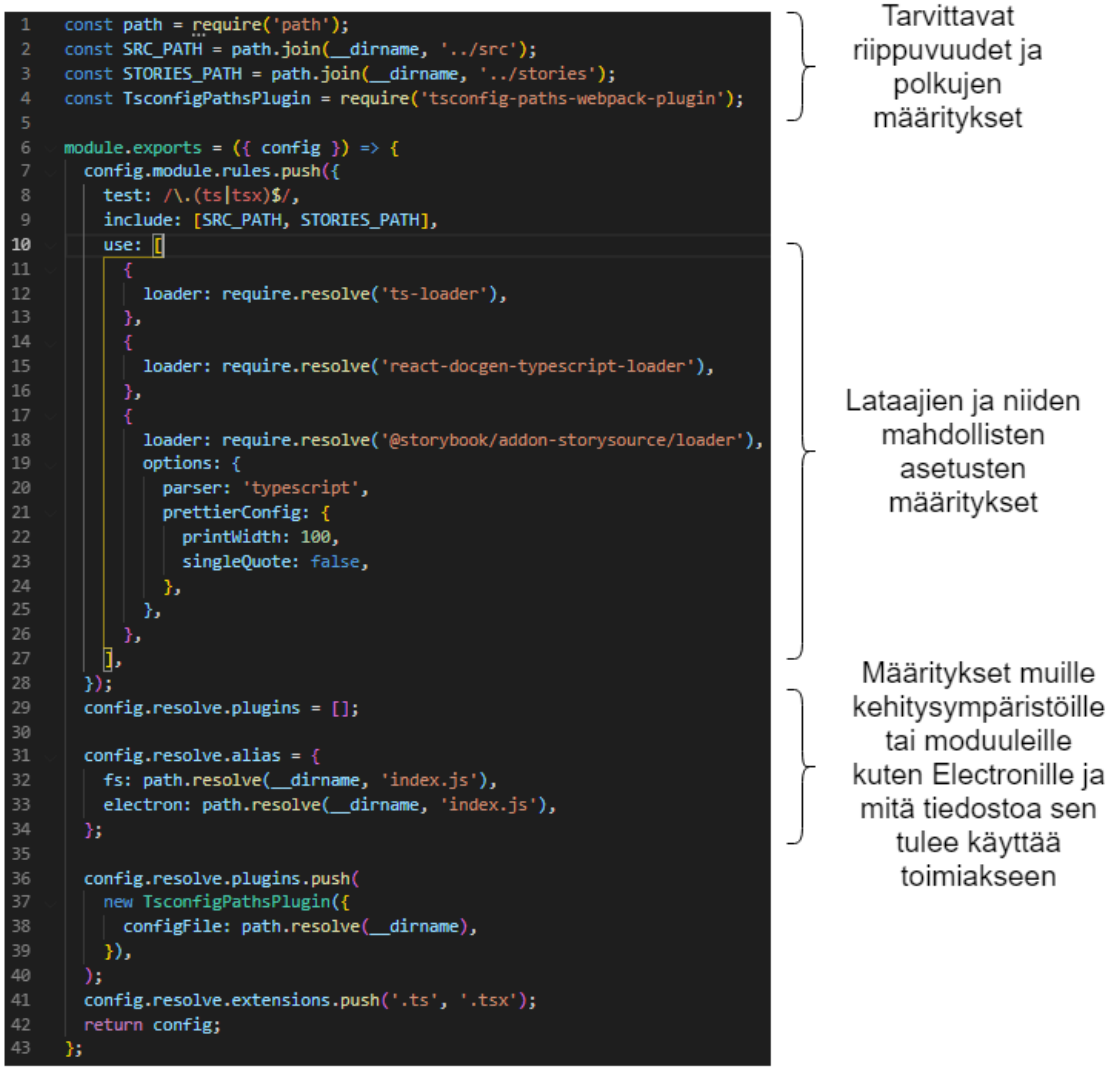
Määrittelytiedostoista tärkeimmät toiminnan kannalta ovat Storybookin, TypeScriptin sekä moduulien paketointimäärittelytiedostot. Ensimmäiseen näistä config.js (kuva 19) tiedostoon määritellään mistä ja minkälaisia tiedostoja Storybookin tulee ladata toimiakseen eli käytännössä tässä tapauksessa *sto-*

ries.tsx -tiedostot. Lisäksi voidaan määritellä muita valinnaisia asetuksia Storybookin toimintaan tai ulkonäköön liittyen, kuten teemoja tai määritellä kaikille tarinoilla, joitakin yhteisiä toiminnallisuuksia.

Kuva 19. Esimerkki config.js tiedoston määrittelystä

TypeScriptin määrittämiseksi tarvitaan erillinen tsconfig.json -tiedosto. Tämä sisältää kaikki tarvittavat määritykset siihen kuinka lopullinen sisältö tulee kääntää käyttöä varten JavaScriptiksi. Siihen voidaan esimerkiksi määrittää mihin JavaScript versioon koodi käännetään tai mistä lähteestä se tarkastelee tyypityksiä jne.

Webpack.config.js -tiedosto on taas staattinen moduulien paketoija JavaScriptille. Lähtökohtaisesti ja riippuen tilanteesta, ei ole tarvetta koskea tähän tiedostoon, mutta tässä tapauksessa, jotta Storybook toimisi oikein, siihen on määritettävä halutut lataajat (loaders) sekä kuinka ottaa huomioon Electron-kehitysympäristö. Esimerkiksi alhaalla olevasta kuvasta (kuva 20) näemme, mitkä tarvittavat lataajat ovat Storybookille tai sen lisäosille tarpeellisia/pakollisia jotta ne toimisivat oikein tai mistä tiedostosta Electron tai muut moduulit hakevat toiminnallisuutensa. Tässä tapauksessa Electronin määrittävä index.js tiedosto on Storybook -kansion juuressa ja se on tyhjä. Sen sisältö tavallaan jäljittelee varsinaista toiminnallisuutta.




```
const path = require('path');
const SRC_PATH = path.join(__dirname, '../src');
const STORIES_PATH = path.join(__dirname, '../stories');
const TsconfigPathsPlugin = require('tsconfig-paths-webpack-plugin');

module.exports = ({ config }) => {
  config.module.rules.push({
    test: /\.(ts|tsx)$/,
    include: [SRC_PATH, STORIES_PATH],
    use: [
      {
        loader: require.resolve('ts-loader'),
      },
      {
        loader: require.resolve('react-docgen-typescript-loader'),
      },
      {
        loader: require.resolve('@storybook/addon-storysource/loader'),
        options: {
          parser: 'typescript',
          prettierConfig: {
            printWidth: 100,
            singleQuote: false,
          },
        },
      },
    ],
  });
  config.resolve.plugins = [];

  config.resolve.alias = {
    fs: path.resolve(__dirname, 'index.js'),
    electron: path.resolve(__dirname, 'index.js'),
  };

  config.resolve.plugins.push(
    new TsconfigPathsPlugin({
      configFile: path.resolve(__dirname),
    }),
  );
  config.resolve.extensions.push('.ts', '.tsx');
  return config;
};
```

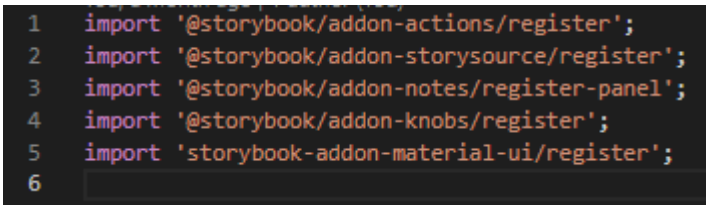
Kuva 20. Esimerkki webpack.config.js -tiedostosta

Nyt on tarvittavat määritykset ja liitännäiset Storybookin perustoiminnallisuuksien käyttöön. Seuraavaksi laajennetaan sen toiminnallisuutta erilaisia lisäosilla ja katsotaan, miten ne tulee määrittää.

5.2 Storybook työympäristön lisäosat ja niiden asennus

ObSAS 2.0 varten Storybookin toiminnallisuuksia ja ominaisuuksia tulee laajentaa muutamalla lisäosalla. Ne eivät ole välttämättömiä, mutta auttavat saavuttaman halutut tavoitteet paremmin sekä laajentavat jo nykyisen projektin olemassa olevien kehitysominaisuuksia pidemmälle, kuten testausympäristöä. Seuraavasta liitteestä (ks. liite 4) nähdään kaikki tarvittavat lisäosat ja niiden komennot tarvittavineen liitännäisineen. Joidenkin lisäosien kohdalla tarvitaan myös lisämäärityksiä muihin määrittelytiedostoihin tai erillisiä tyypityksiä, jos niille ei ole sellaisia määritetty kehittäjien toimesta. Kaikki lisäosat kuten Storybook itsekin, asennetaan kehittäjien liitännäisiin.

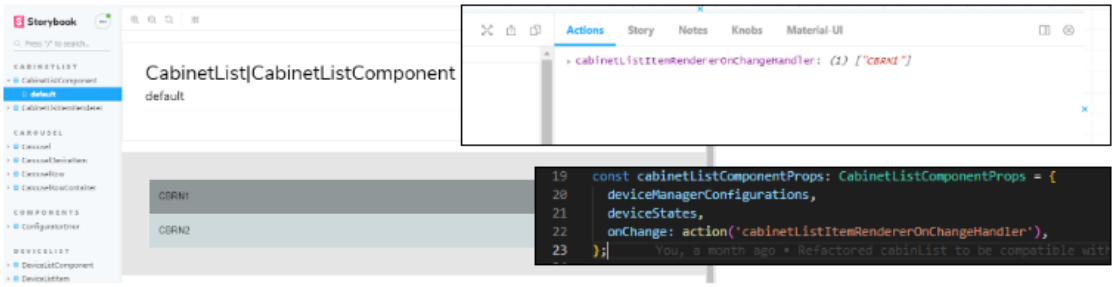
Jotta asennetut lisäosat tulisi käyttöön, on niitä varten tehtävä muutamia määrityksiä. Yksi tämänlainen on addons.js -tiedosto, johon määritellään kaikki Storybookin addon -paneelissa näkyvät lisäosat (kuva 21). On myös mahdollista, ettei kaikki asennettuja lisäosia tarvitse määrittää tähän tiedostoon, koska niiden toiminnallisuus ei vaadi sitä tai ne toimivat jonkin toisen lisäosan kautta. Tässä tapauksessa Addon Docgen -lisäosa on yksi tämänlaisesta esimerkistä sen toimiessa Addon Info -lisäosan kautta.



Kuva 21. Addon.js -määrittelytiedosto.

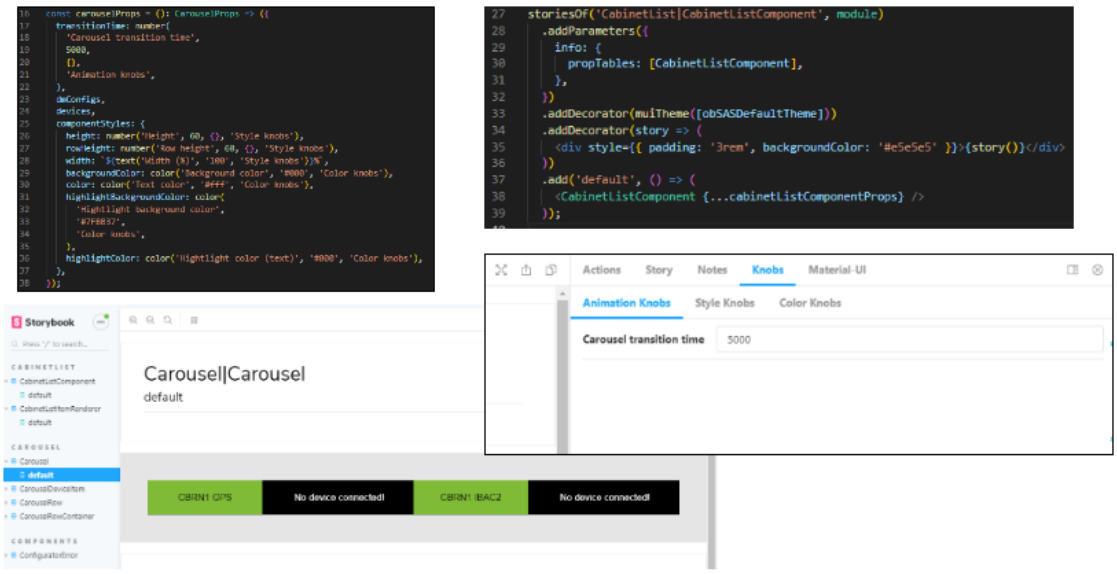
Toiminnallisuus, jonka yksittäinen lisäosa tuo, vaihtelee paljon. Osa näistä on enemmän toiminnallisia ja osa puolestaan dokumentointia auttavia lisäosia. Lisäosat antavat myös mahdollisuuden testata komponenttien toiminnallisuutta enemmänkin, kun käyttäjä voi itse vaikuttaa käytettävään jäljittelytietoon. Asennetuista lisäosista toiminnallisia ovat mm. Action -addon, Knobs -addon ja Material UI Theme -addon ja dokumentoivia Notes -, Info - ja Storysource -lisäosat. Testaukseen on myös omat lisäosat, joita käsitellään testausympäristöosiossa enemmän.

Näistä ensimmäinen, Action addon auttaa jäljittelemään ja esittämään tapahtumien (events) tietoa. Tätä varten *stories.tsx* -tiedostoon tulee tuoda action -toiminto ja määritellä sen paikka sekä mitä sen halutaan palauttavan. Esimerkkikuvassa (kuva 22), määritellään action -toiminnon paikka onChange -kuuntelijafunktioon ja halutaan sen palauttavan merkkijono, kun tapahtuma toteutuu. Palautettu tieto on tarkasteltavissa Action -välilehdeltä Addon -paneelissa. (GitHub 2019b.)



Kuva 22. Esimerkki Action -lisäosan toiminnasta.

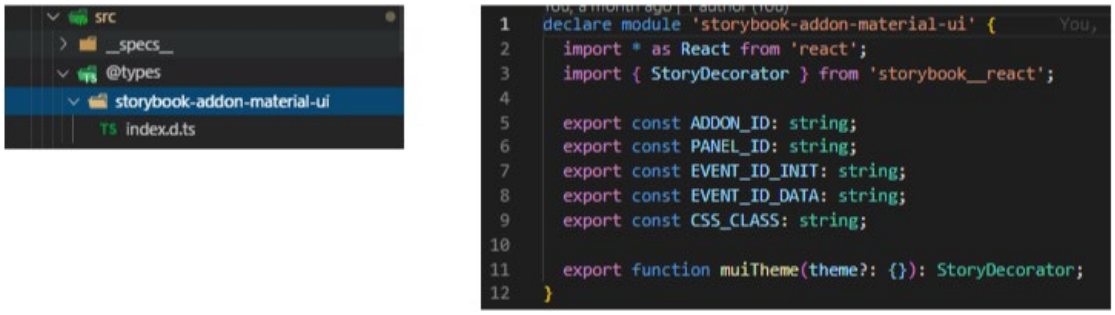
Knobs -lisäosa antaa mahdollisuuden muokata komponentin ominaisuuksien sisältöä dynaamisesti Storybook -näkymästä sen ollessa käynnissä (kuva 23). Perustoiminnallisuus tapahtuu tuomalla komponentin ominaisuutta vastaava funktio, eräänlaisia knobs -funktioita. Niitä voi olla esimerkiksi *number()*, *text()*, *color()* jne, joista jokainen ottaa vastaan ja palauttaa tietyn muotoista/tyyppistä tietoa. Näihin funktioihin voidaan sitten määritellä otsikko, oletusarvo, mahdolliset lisäasetukset ja mahdollinen alaotsikko, joka näkyy välilehtenä addon -paneelissa Knobs -välilehden alla. (GitHub, 2019c.) Koska usein syötetty tieto on objektimuotoista ja se saattaa sisältää sisäkkäisiä objekteja, on hyvä määritellä funktio, joka palauttaa syötettävän tiedon tarinoille. Näin pystytään sisäkkäisiin objekteihin määrittelemään knobs -funktioita. Jotta knobs saadaan käyttöön tarinoiden sisällä, täytyy niihin määritellä tarinakohtaisesti tai yhteisesti addDecorator(), joka saa withKnobs -parametrin (GitHub 2019c).



Kuva 23. Esimerkki Knobs -lisäosan toiminnasta

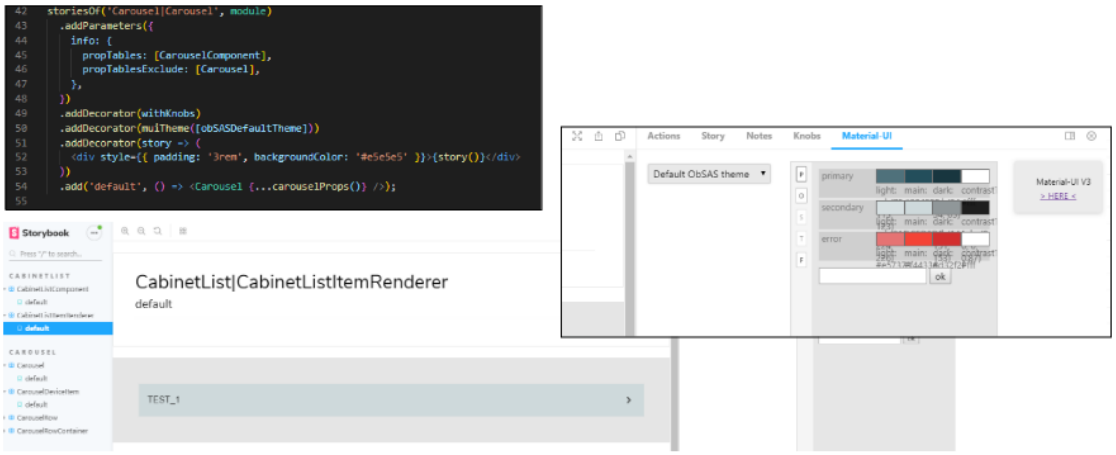
Material UI Theme -lisäosa on tehty auttamaan Material UI -komponenttien luomista. Se mahdollistaa teemojen testaamisen ja luomisen sekä antaa mahdollisuuden dynaamisesta vaihtaa teemojen väriarvoja Storybookin ollessa

käynnissä (GitHub 2019d). Se on vielä alpha -kehitysvaiheessa ja tämän vuoksi siihen ei ole olemassa kehittäjien tarjoamia tyypityksiä TypeScriptiä varten, vaan ne on määriteltävä itse (kuva 24),



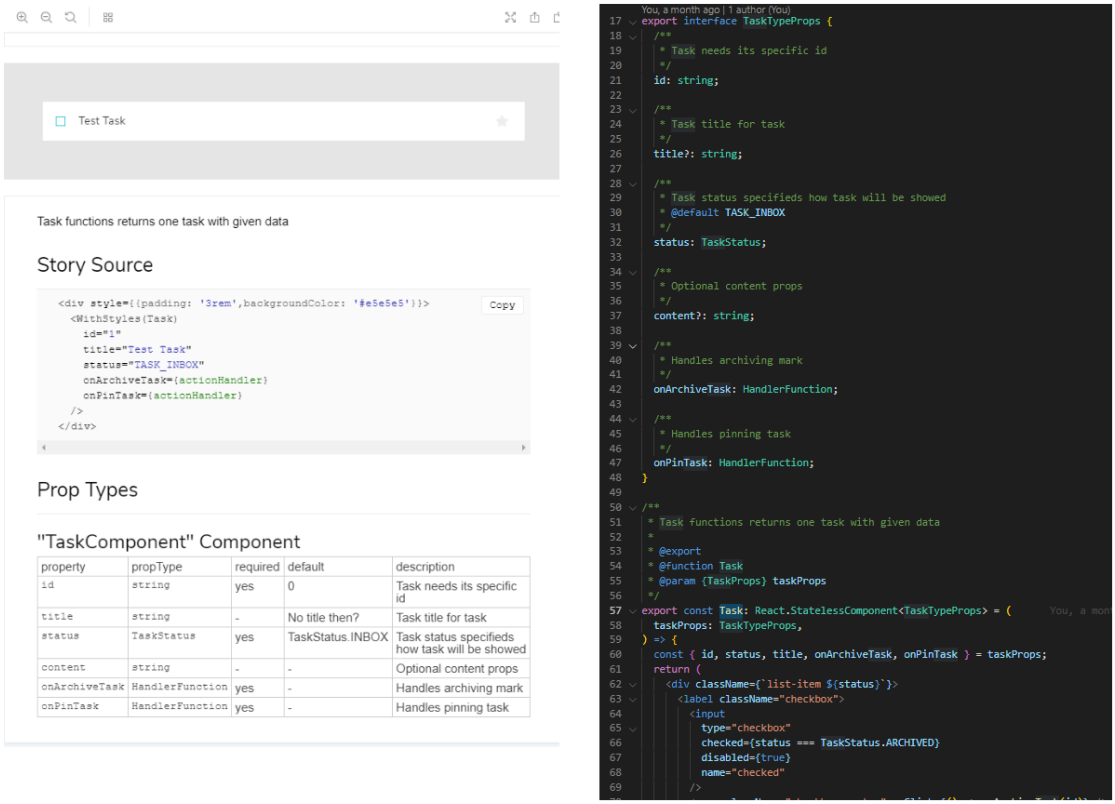
Kuva 24. Material UI Theme -lisäosan tyypitysten määrittelystä

eikä lisäosa välttämättä kuten ObSAS 2.0 kohdalla toimi visuaalisesti oikein. Kuten aikaisemminkin lisäosa otetaan käyttöön addDecorator() avulla, joka ottaa nyt vastaan muiTheme() -funktion, jonka parametrina on lista (array) teemaobjekteista. Alla oleva kuva (kuva 25) näyttää lisäosan toiminnallisuutta, josta voidaan huomata, ettei lisäosan ulkoasu toimi täydellisesti.



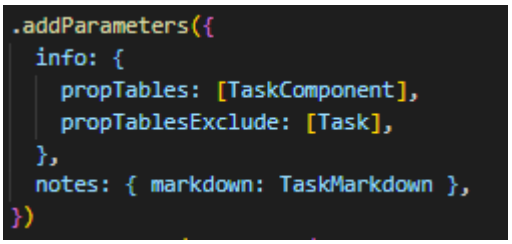
Kuva 25. Esimerkki Material UI Theme -lisäosasto

Dokumentaatioita auttavia lisäosista Info -, yhdessä Docgen -lisäosan kanssa mahdollistavat JavaScript Dokumentaatioista (JSDoc) muodostettavan dokumentoinnin lisäämisen Storybookin tarinoihin automaattisesti (kuva 26). Näin pystytään esimerkiksi komponenttiin syötettävistä tiedoista muodostamaan automaattinen taulukko, josta nähdään kaikki tarvittava syötettävään tietoon liittyvä tieto esim. onko se pakollinen tai siihen lisätty kuvaus (GitHub 2019e). Info -lisäosan avulla pystytään myös esittämään käyttäjän itse määrittämää tietoa esimerkiksikomponenttiin liittyen. Info -lisäosan ominaisuuksia pystytään myös määrittämään ja muokkaamaan pitemmälle addParameters() ominaisuuden kautta (GitHub 2019f).



Kuva 26 Esimerkki Info -ja Docgen -lisäosan toiminnasta

Notes -lisäosa mahdollistaa käyttäjän määrittämien muistiinpanojen tai vastaavanlaisen lisäämisen tarinoihin. Tämä tulee näkyviin lisäosa -paneelin Notes -välilehdelle. Määritettävä tieto voi olla tekstiä tai se voi myös olla erillinen markdown -tiedosto, johon voidaan markdown -syntaksia käyttämällä tehdä halutut merkinnät yms. (NPM 2019.) Varsinaisesti käyttöön saatavaksi, tulee Notes -lisäosan sisältö määritellä addParameters() avulla (Kuva 27).



Kuva 27. Esimerkki Notes -lisäosan määrittämisestä

Storysource -lisäosan avulla pystytään näkemään Storybook sisällä käyttäjän luomat tarinat koodeineen Story -välilehdeltä lisäosa -paneelista. Se myös näyttää mikä tarina on milläkin hetkellä auki käyttäjän tarkasteltavana. Käyttäjä voi tarvittaessa siirtyä tämän lisäosan avulla tiettyä tarinan koodiosiota klikkaamalla halutun tarinan tarkasteluun. Poiketen muiden lisäosien määrittelystä, Storysource -lisäosa määritellään webpack.config.js -tiedostossa ja tätä

varten se tarvitsee myös sopivan lataajan. Tässä tapauksessa storysource-loader. (GitHub, 2019g.) Seuraavasta kuvasta (kuva 28) näemme miltä story-source-loader määritys näyttää.

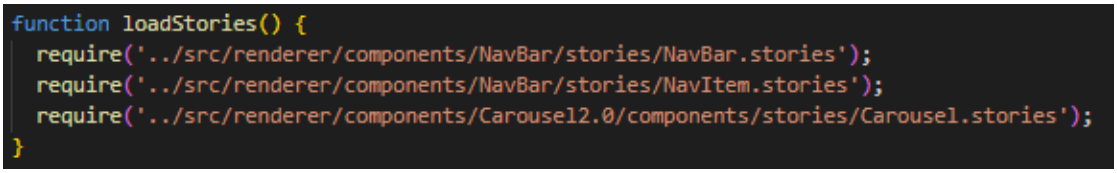
Kuva 28. Esimerkki storysource-loader määrittämisestä

Nyt on lähes kaikki ObSAS 2.0 varten tarvittavat lisäosat asennettu ja määritetty. Testiosiossa tarkastellaan vielä testaukseen tarvittavien lisäosien määrittäminen ja käyttö.

5.3 Storybook ja testausympäristön käyttö

Koska ObSAS 2.0 ja Observis Oy yleensäkin on omaksunut testausympäristön osaksi omaa kehitystyötään, on myös suotavaa katsoa mitä Storybook voi osaltaan tähän tuoda lisää. Testausympäristöä varten asennetaan ObSAS 2.0 -projektiin vielä, jo aikaisemmin Lisäosat -luvussa mainitut StoryShots -ja Storyshots-Puppeteer -lisäosat, jotka molemmat tarvitsevat toimiakseen Jest-testausympäristöä.

Näistä StoryShots tarvitsee toimiakseen siis Jest-testausympäristöä, joka on asennettu jo ennestään ObSAS 2.0 -projektiin. Koska Jest ei itsessään ymmärrä require.context() -määritystä, jolla tarinat ladataan sen ollessa yksinomaan webpack -toiminto. Toisekseen projekti ei käytä Babel -kääntäjää, niin lisäosan dokumentaatioiden tarjoama ratkaisu tähän ongelmaan ei toimi. Ainoa tapa saada nyt StoryShots -lisäosa ja tarinat toimimaan yhdessä, on tuoda ne yksitellen config.js -määrittely tiedostoon (kuva 29) lataamista varten tai vaihtoehtoisesti asentaa Babel ja käyttää lisäosan dokumentaation ohjeita.



Kuva 29. Tarinoiden määrittely StoryShots -lisäosaa varten

Kun nämä edellä mainitut määritykset ja muutokset on tehty, voidaan luoda uusi kansio Jest-testausympäristön määritysten mukaisesti, jotta sen sisällä olevat spec.ts/tsx -tiedostot voidaan ladata testejä ajettaessa. Kansion sisään luodaan siis spec.ts/tsx -tiedosto, johon määritellään Snapshot -testaus tarinoille, mahdolliset asetukset sekä toiminto, että se tehdään kaikille tarinoille (kuva 30). Kun testit ajetaan, otetaan jokaisesta tarinasta storyshots -tiedosto, joka kuvaa tarinan rakenteen.

```
import initStoryshots, {
  multiSnapshotWithOptions,
} from '@storybook/addon-storyshots';
import * as path from 'path';

initStoryshots({
  integrityOptions: { cwd: path.join(__dirname, '..', 'stories') },
  test: multiSnapshotWithOptions({}),
});
```

Kuva 30. Malli Storyshots snapshot -testin määrityksestä

Storyshots-Puppeteer on puolestaan visuaalista testausta varten. Kuten aikaisemmin StoryShots -lisäosan kohdalla, myös tämä lisäosa vaatii Jest-testausympäristöä toimiakseen sekä erityistä jest-image-snapshot -liitännäistä, jota ei tarvitse kuitenkaan asentaa erikseen, koska se tulee lisäosan asennuksen yhteydessä. Samoin kuin aikaisemmin voidaan määritellä erillinen spec.ts/tsx -tiedosto (kuva 31), johon määritellään -toiminnon asetukset ja URL -polku, mihin se kohdistetaan. Polku voi johtaa paikalliseen koontiversioon, julkaistuun versioon tai polkuun, jossa kehityksessä oleva Storybook on käynnissä.

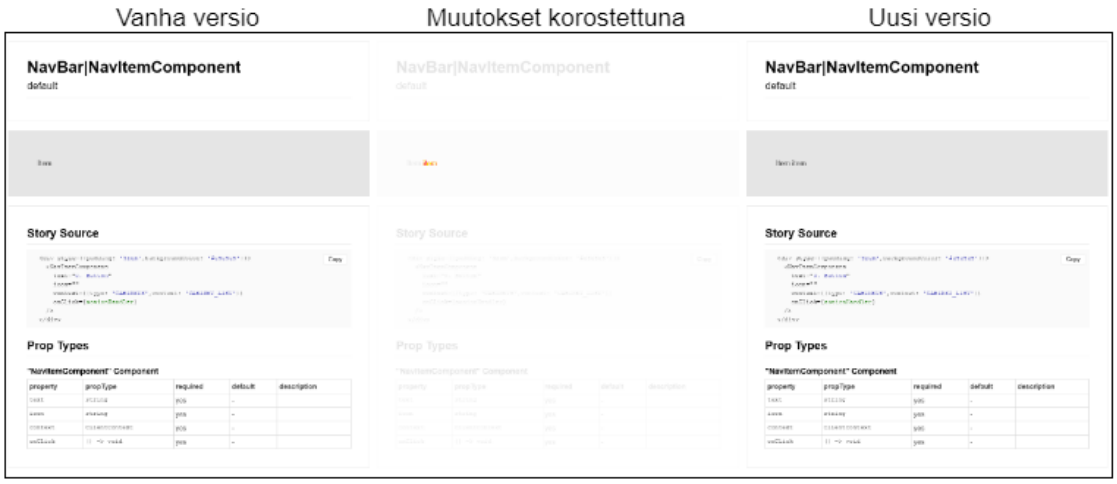
```
import initStoryshots from '@storybook/addon-storyshots';
import { imageSnapshot } from '@storybook/addon-storyshots-puppeteer';

const getScreenshotOptions = () => {
  return {
    fullPage: true,
  };
};

initStoryshots({
  suite: 'Image storyshots',
  test: imageSnapshot({
    storybookUrl: 'http://localhost:9001/',
    getScreenshotOptions,
  }),
});
```

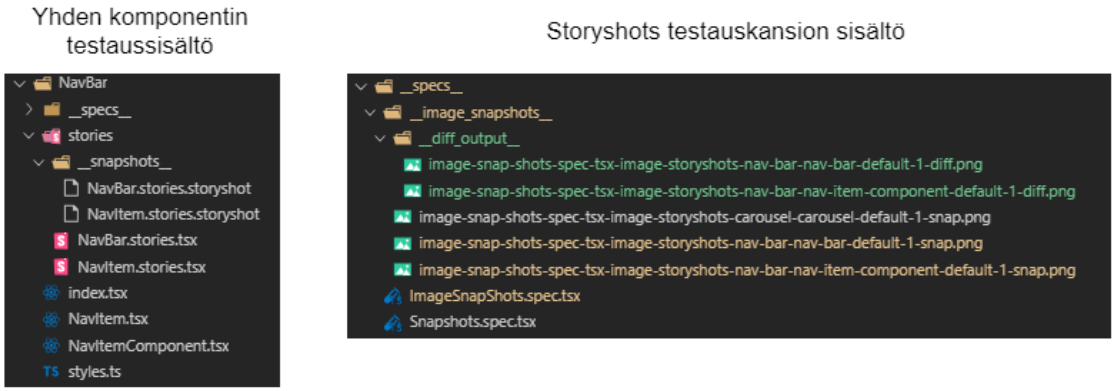
Kuva 31. ImageSnapshot -toiminnon määrittäminen

Kun testit tämän jälkeen ajetaan, jokaisesta tarinasta otetaan kuva ja se tallennetaan snapshot -testauksen tavoin talteen. Jos komponenttiin tai tarinan sisältöön tehdään muutoksia ja testit ajetaan uudestaan, muodostaa lisäosa vertailun uuden ja vanhan kuvan (kuva 32) väliltä osoittaen korostevärille paikat mihin muutokset kohdistuvat. Muutokset voidaan kumota tai vastaavasti hyväksyä, jolloin vanha versio korvataan uudella.



Kuva 32. ImageSnapshot kuvien vertailu

Kaikkien edellä mainittujen määritysten jälkeen, pitäisi kansiorakenteiden näyttää yhden komponentin ja Storyshots -testikansion osalta esimerkiksi alla olevan kuvan (kuva 33) mukaiselta.

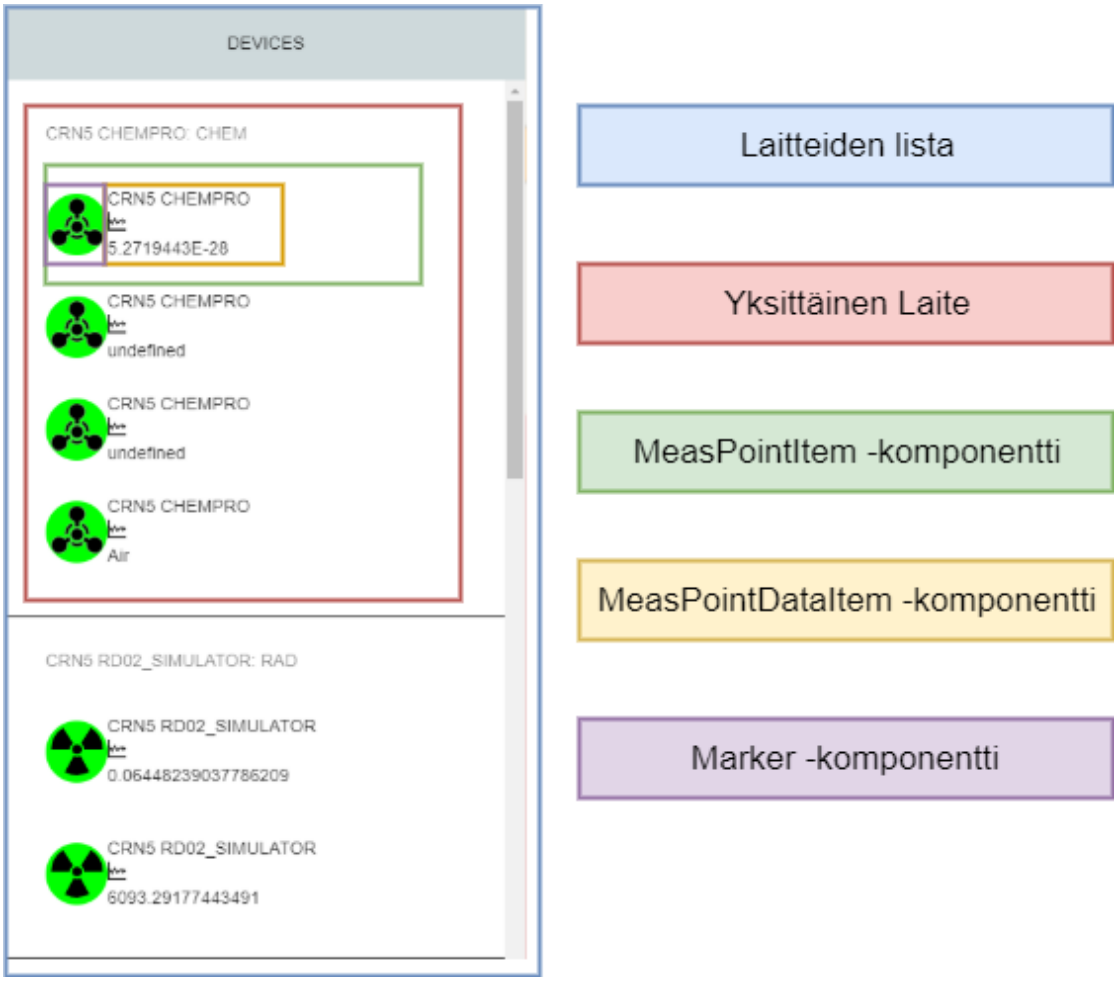


Kuva 33. Kansioiden rakenne ja sisältö

Näillä määrityksellä pitäisi nyt olla projektiin sopiva testausympäristö. On hyvä myös tiedostaa, että StoryShots ja Storyshots-Puppeteer -lisäosille voidaan määritellä muitakin asetuksia, jotka vaikuttavat siihen, kuinka testaus tapahtuu. ObSAS 2.0 -projektin kannalta toistaiseksi niille ei ole tarvetta.

5.4 Esimerkkikomponentti

Seuraava esimerkkikomponentti näyttää kuinka komponenttilähtöisen kehitys toimii Reactia ja sen eri komponentti suunnittelumalleja käyttäen ObSAS 2.0 -projektissa. Esimerkkikomponenttina käytetään jo aikaisemmin luotua laitelista (Devicelist) -komponenttia (kuva 34), joka on koostettu pienemmistä yksittäisistä komponenteista. Samalla käytetään rakennettua Storybook-työympäristöä sen eri tilojen esittämiseen ja komponentin dokumentoimiseen.



Kuva 34. Esimerkkikomponentti ja sen rakenne.

Koska ObSAS 2.0 on graafisten käyttöliittymien luomiseen tehty työkalu, tulee sen sisällä käytettävien käyttöliittymäkomponenttien toimia useassa paikassa omina erillisinä komponentteina ilman muista komponenteista aiheutuvia riippuvaisuuksia. Tämän takia lähes jokainen käyttöliittymäkomponentti tai sen osa on tällä hetkellä erityisen riippuvainen sovelluksen tilasta eli tässä tapauksessa Redux -kirjastosta. Storybookin osalta tämä aiheuttaa tarpeen saada sovelluksen tila komponenttien käyttöön, jotta ne toimisivat oikein.

Yksinkertaisuudessaan mikäli Redux -kirjasto on asennettu ja määritelty oikein projektiin ja kaikki tyypitykset täsmäävät, kun sen funktioita käytetään, jokainen Storybook -avulla kuvatun yksittäisen komponentin sisällä oleva tarina ympäröidään Provider -komponentilla, jolle annetaan ominaisuutena store -objektin, mutta koska ObSAS 2.0 on määritelty Redux -kirjaston toiminnallisuutta enemmänkin, tämä ratkaisu ei suoraan toimi, vaan antaa tyyppivirheitä. Ratkaisuna tähän on luoda funktio (ks. liite 5), joka palauttaa aina halutunlaisen store -objektin jokaiseen tarinassa kuvattuun tilaan tai tilanteeseen Provider -komponentille (kuva 35). Tämä myös antaa liikkumavaraa mitä tietoa jäljitellyn sovelluksen tilan ja store -objektin halutaan sisältävän. Tämä funktio saa parametreina halutut reducer:it ja halutun sovelluksen tilan sisällön. Tyypitysten avulla voidaan määritellä, mitkä reducer:it ovat käyttökelpoisia parametreina funktion sisällä.

```
.add('Default', () => (
  <Provider
    store={createMockStore(
      {
        deviceState: reducers.deviceStateReducer,
        contextState: reducers.contextReducer,
        deviceInfoProvider: reducers.deviceInfoReducer,
      },
      {
        contextState: defaultContextState,      You, 12 days ago • Change De
        deviceInfoProvider: defaultDeviceInfoProvider,
        deviceState: defaultDeviceState,
      },
    )}
  >
    <DeviceList.descriptor.componentClass {...deviceListComponentProps()} />
  </Provider>
))
```

Kuva 35. Esimerkki Redux -store:n jäljittelyfunktion käytöstä

Redux -kirjaston määrittämisen jälkeen voidaan siirtyä varsinaiseen tilojen kuvaamiseen. Järkevintä tämänlaisessa tilanteessa ja yleensäkin storybook-lähtöisessä kehityksessä on alkaa sovelluksen kehittäminen ja komponenttien tilojen kuvaaminen koostetun komponentin pienimmästä ja alimmaisemmasta komponentista. Tässä tapauksessa ne ovat Marker -komponentti ja MeasPointDataItem -komponentti. Marker -komponentti on koostettu komponentti ja ottaen huomioon, ettei kaikille tämän komponentin sisällä oleville toiminnoille ole käyttöä Laitelista -komponentissa, voisi koko komponentin sijasta käyttää MarkerImageComponent -komponenttia tai vaihtoehtoisesti muuta ratkaisua, joka sisältää vain kuvan etsimiseen ja näyttämiseen tarvittavat toiminnot. Joka

tapauksessa kuvataan Storybookin avulla kyseisen komponentin tilat, jotka ovat relevantteja Laitelista -komponenttia varten, eli ainakin laitteiden tilojen värit ja mahdolliset eri kuvat, joita komponentti voi käyttää. Koska komponentti on riippuvainen Redux -kirjastosta, tarjotaan sille Provider -komponentti ja tarvittavia tietoja vastaavan tilan tiedot. MeasPointDataItem -komponentti on puolestaan yksinkertaisempi ja eikä se ole riippuvainen Redux -kirjastosta. Sen tilat voidaan kuvata suoraan määrittämällä tarvittavat komponentin ominaisuudet, jotka toteuttavat halutun tilan.

Seuraavana komponenttina järjestyksessä on MeasPointItem -komponentti sekä siitä vielä ylempänä DeviceItem -komponentti. Nämä ovat hyvä esimerkki, siitä kuinka tärkeä Redux -kirjasto on ObSAS 2.0 -projektissa. Koska kyseiset komponentit eivät käytä itsessään Redux -kirjastoa, mutta sen alhaalla olevat komponentit käyttävät, on se määriteltävä kuten aikaisemminkin Storybook tilojen kuvausta varten tarvittavilla tiedoilla.

Viimeisimpänä on siis koko Laitelista -komponentin kuvaaminen Storybookilla. Nyt on hyvä pohtia, onko tarvetta kuvata kaikki tilat erikseen, jotka aikaisemmissa komponenteissa alikomponenteissa kuvattiin näkyville myös tämän komponentin kohdalla. Se vaatisi paljon eri tiloja ja jäljittelytieto eri muodossa. Tärkeintä tässä vaiheessa olisi, että komponentin pääasiallinen toiminto tulisi ainakin näkyviin ja aikaisemmin kuvatut komponentit hoitavat muun.

Edellä mainituille sekä lähtökohtaisesti kaikille komponenteille, joille voidaan määritellä ominaisuuksia (probs), olisi hyvä myöskin jsDoc dokumentointitavan mukaisesti kuvata lyhyesti mitä varten kyseinen ominaisuus on. Nämä dokumentoinnit tulevat lopulta näkyviin Storybook -työnäkymässä.

5.5 Lopputulos ja jatkokehitys

Koska ObSAS 2.0 on vielä kehityksessä ja varsinkin alkuvaiheessa, siinä etsitään koko ajan erilaisia kehitystä eteenpäin vieviä ratkaisuja, mitkä tukisivat ja nopeuttaisivat kehitystyötä. Tämän vuoksi useat komponentit eroavat ratkaisuiltaan toisistaan paljonkin. Esimerkiksi tyylien määrittely vaihtelee styled -komponenteista styles.ts -tyylitiedosto määrittelyyn tai sitten kansiorakenne on

aivan erilainen kuin toisissa. Tähän kun lisätään vielä tarve esittää ja dokumentoida komponentit Storybookin avulla, joka tarvitsee ObSAS 2.0 kohdalla myös Redux -kirjastoa toimiakseen, ongelmia ja ratkaistavaa syntyy hyvin suurella todennäköisyydellä. Oman haasteen tuo myös Electron-kehitystyöympäristö, jonka omat toiminnallisuudet edellyttävät sen toimintojen jäljittelyä Storybookia varten. Lopputuloksena voidaan kyllä sanoa, että Storybook kyllä toimii projektissa, mutta jotta se tehostaisi kehitystyötä halutussa määrin, tulee muutaman asian muuttua ennen sitä. Nämä toimivat samalla jatkokehitysehdotuksina

Ensimmäinen jatkokehitysehdotus on yhtenäistää koodauskäytännöt ja sopia mitkä ovat kehitystyön kannalta ne asiat, jotka täytyy tehdä tietyllä tavalla aina poikkeuksitta. Näitä voisivat olla kansiorakenteiden määrittely, tavat kuinka komponentit luodaan (komponenttilähtöinen kehitys) tai tyylien määritykset. Tämä tehostaisi myös Storybookin avulla työskentelyä, kun kaikki komponentit ovat rakenteellisesti samanlaisia ja helpommin tulkittavia. Koodauskäytäntöjen tutkiminen kehitystyössä voisi olla myös tulevaisuudessa yksi Observiksen opinnäyte – tai tutkimustyöaihe.

Toinen jatkokehitys liittyy siihen, kuinka jäljittelytietoa saadaan ja voitaisiin käyttää tehokkaammin. Tämä liittyy aivan siihen tarpeeseen, että ObSAS 2.0 ja sen komponentit saavat valtavasti erilaista ja erimuotoista tietoa. Tämän tiedon hallitseminen ja varsinkin tilojen tai yhden tilan kuvaaminen yhtä tai useampaa tiedon osaa vaihtaen ja vielä oikean lopputuloksen saaminen on hankalaa nykyisessä muodossaan. Tämä voisi toimia myöskin tulevaisuudessa yhtä opinnäyte – tai tutkimustyöaiheena.

Viimeisenä jatkokehitysehdotuksena olisi tutkia kuinka React Hook -toiminnot voisivat auttaa sovelluksen tilan hallintaa Redux -kirjaston ohessa ja helpottaisiko se kehitystyötä Storybookilla ja yleensäkin kehitystyötä nykyisestään komponenttien parissa.

Kaikki edellä mainitut jatkokehitysehdotukset on myös Observis Oy:llä tiedostettu ja siellä katsotaankin niiden olevan kehityksen kannalta tärkeitä kohteita tulevaisuudessa. On hyvinkin todennäköistä, että ainakin ensimmäinen ja toinen ehdotus tulevat olemaan opinnäytetyön aiheita jollakin muotoa tai ainakin

osana jotakin isompaa aihekokonaisuutta. Viimeinen liittyen React Hook -ominaisuuden tutkimiseen on jo aloitettu yleisellä komponenttikehityksen tasolla.

6 PÄÄTÄNTÖ

Nyt kun kaikki aiheet teoriasta ja käytännöstä on käyty läpi, voin kirjoittajana sanoa olevani tyytyväinen. Toteutusaika opinnäytetyölle oli pitkä ja työstämisen aikana vastaan tuli paljon sellaisia ongelmia, joihin ratkaisun löytäminen vei aikaa, johtuen osittain Observis Oy:n alkuperäisestä tavoitteesta. Myös oma jaksaminen työstää opinnäytetyötä normaalin työssäkäynnin ohella vaikutti työn edistymiseen.

Ongelmia, joita työskentelyn aikana erityisesti ilmeni, oli muun muassa toteutustapojen löytäminen Electron- ja Redux-ympäristöille Storybookin kanssa, joista erityisesti Reduxille alkuperäinen ajatus oli pitää se erillään Storybookista. Lopulta kuitenkin ilman selkeää toimivaa ratkaisua alkuperäiseen tavoitteeseen, päädyttiin ottamaan Redux osaksi Storybook-kehitystä. Näin saatiin kohtuu hyvin toimiva kehitysympäristö.

Observis Oy on ollut kuitenkin tyytyväinen lopputulokseen, vaikkakin Storybook ei täydellisesti sovi ObSAS 2.0 kehitykseen projektin nykyisessä muodossa. Tästä erityisesti huomioiden tuli esiin ne seikat, missä tulisi yritystä kehittää edelleen paremmaksi, jotta myös komponenttikehitys Storybookia käyttämällä toimisi halutulla tavalla. Myös esille tuomani kehitysehdotukset on huomioitu ja tulevat suurella todennäköisyydelle hyvinkin olemaan opinnäytetyön aiheita lähitulevaisuudessa. On myös todennäköistä, että Storybook voitaisiin ottaa osaksi jotain perinteisempää verkkosovelluksen kehitystä yrityksessä.

Varsinainen työskentely aiheen parissa oli kaikesta huolimatta mielenkiintoista. Aihe, tavoitteisiin nähden oli todella laaja ja vaati jonkin verran kohdentamista, jotta lopullinen työn pituus säilyisi sopivissa rajoissa sekä sisältö eheänä. Uskon, että opinnäytetyö on tässä mielessä onnistunut, vaikka aihetta voisi vieläkin jatkokehittää edelleen pitemmälle ja entistä yksityiskohtai-

sempaan läpikäymiseen. Itse näen vielä hyödyntäväni tutkimaani asiaa mahdollisissa omissa tai muiden yritysten projekteissa, mikäli tilanne antaa siihen mahdollisuuden.

LÄHTEET

Bertoli, M. 2017. React Design Patterns and Best Practices. Birmingham: Packt Publishing Ltd. [viitattu 23.5.2019].

Cox, B. J. 1986. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley. [viitattu 26.6.2019].

Brockhoff, R. 2018. Using Electron with React: The Basics. WWW-dokumentti. Saatavissa: https://medium.com/@brockhoff/using-electron-with-react-the-basics-e93f9761f86f [viitattu 3.9.2019]

Coleman, T. 2017. Component-Driven Development. WWW-dokumentti. Saatavissa:https://blog.hichroma.com/component-driven-development-ce1109d56c8e [viitattu 20.8.2019].

De La Cuadra, A. 2016. Designing Modular UI Systems Via Style Guide-Driven Development. WWW-dokumentti. Saatavissa: https://www.smashingmagazine.com/2016/06/designing-modular-ui-systems-via-style-guide-driven-development/ [viitattu 20.8.2019].

GeegsforGeeks. 2019a. Software Engineering | Coupling and Cohesion. WWW-dokumentti. Saatavissa: https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/ [viitattu 27.6.2019].

GeeksforGeeks. 2019b. ReactJS | State in React. WWW-dokumentti. Saatavissa: https://www.geeksforgeeks.org/reactjs-state-react/ [viitattu 2.6.2019].

Gift, E 2019. An Introduction to Storybook: Organize How You Build JS Components. WWW-dokumentti. Saatavissa: https://scotch.io/tutorials/an-introduction-to-storybook-organize-how-you-build-js-components [viitattu 28.8.2019].

GitHub. 2019a. Specifications Addon. WWW-dokumentti. Saatavissa: https://github.com/mthuret/storybook-addon-specifications [viitattu 30.8.2019].

GitHub. 2019b. Storybook Addon Actions. WWW-dokumentti. Saatavissa: https://github.com/storybookjs/storybook/tree/master/addons/actions [viitattu 28.10.2019].

GitHub. 2019c. Storybook Addon Knobs. WWW-dokumentti. Saatavissa: https://github.com/storybookjs/storybook/tree/master/addons/knobs [viitattu 28.10.2019].
GitHub. 2019d. Storybook Addon Material-UI. WWW-dokumentti. Saatavissa: https://github.com/react-theming/storybook-addon-material-ui [viitattu 28.10.2019].

GitHub. 2019e. Storybook-addon-react-docgen. WWW-dokumentti. Saatavissa: https://github.com/hipstersmoothie/storybook-addon-react-docgen [viitattu 28.10.2019].

GitHub. 2019f. Storybook Info Addon. WWW-dokumentti. Saatavissa: https://github.com/storybookjs/storybook/tree/master/addons/info [viitattu 28.10.2019].

GitHub. 2019g. Storybook Storysource Addon. WWW-dokumentti. Saatavissa: https://github.com/storybookjs/storybook/tree/master/addons/storysource [viitattu 28.10.2019].

Hermel, Z. 2013. Facebook's React JavaScript User Interfaces Library Reaceives Mixed Reviews. WWW-dokumentti. Saatavissa: https://www.infoq.com/news/2013/06/facebook-react/ [viitattu viitattu 27.6.2019].

Hsu, H. 2017. A Short History of Objective-C. WWW-dokumentti. Saatavissa:https://medium.com/chmcore/a-short-history-of-objective-c-aff9d2bde8dd [viitattu viitattu 23.8.2019].

Kozhukharenko, N. 2017. Component driven development: How to guide. WWW-dokumentti. Saatavissa: https://www.slideshare.net/nikolaykozhuharenko/component-driven-development-how-to-guide [viitattu 5.9.2019].

Naur, P & Randell, B. 1969. Software engineering. WWW-dokumentti. Saatavissa: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF [viitattu 23.8.2019].

Nguyen, D. 2017. UI component explorers – your new favorite tool. WWW-dokumentti. Saatavissa: https://blog.hichroma.com/the-crucial-tool-for-modern-frontend-engineers-fb849b06187a [viitattu 29.8.2019].

NPM. 2019. Storybook Addon Notes. WWW-dokumentti. Saatavissa: https://www.npmjs.com/package/@storybook/addon-notes [viitattu 28.10.2019].

Ravivhandran, A. 2019. React – Composition VS. Inheritance WWW-dokumentti. Saatavissa: https://programmingwithmosh.com/react/react-composition-vs-inheritance/ [viitattu. 28.8.2019].

React. 2017. React v15.5.0. Blogi. Saatavissa: https://reactjs.org/blog/2017/04/07/react-v15.5.0.html [viitattu 9.7.2019].

React. 2019a. WWW-dokumentti. Saatavissa: https://facebook.github.io/create-react-app/docs/adding-typescript [viitattu 27.6.2019].

React. 2019b. Typechecking With PropTypes. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/typechecking-with-proptypes.html [viitattu 21.7.2019].

React. 2019c. Components and Props. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/components-and-props.html [viitattu 2.6.2019].

React. 2019d. Component State. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/faq-state.html [viitattu 2.6.2019].

React. 2019e. The Component Lifecycle. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/react-component.html#shouldcomponentupdate [viitattu 21.7.2019].

React. 2019f. ComponentDidMount(). WWW-dokumentti. Saatavissa: https://reactjs.org/docs/react-component.html#componentdidmount [viitattu 21.7.2019].

React. 2019g. Using the State Hook. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/hooks-state.html [viitattu 28.7.2019].

React. 2019h. Composition vs Inheritance. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/composition-vs-inheritance.html [viitattu 28.8.2019].

React. 2019i. Higher-Order Components. WWW-dokumentti. Saatavissa: https://reactjs.org/docs/higher-order-components.html [viitattu 5.9.2019].

Redux. 2019. Getting Started with Redux. WWW-dokumentti. Saatavissa: https://redux.js.org/introduction/getting-started [viitattu 26.8.2019].

Rouse, M. 2016. Component. WWW-dokumentti. Saatavissa: https://whatis.techtarget.com/definition/component [viitattu 3.9.2019].

Sairing, J. 2019. A Guide to Component Driven Development. WWW-dokumentti. Saatavissa: https://itnext.io/a-guide-to-component-driven-development-cdd-1516f65d8b55 [viitattu 21.8.2019].
Schwartz, N. Storybook Driven Development 2017. WWW-dokumentti. Saatavissa: https://medium.com/nulogy/storybook-driven-development-a3c517276c07 [viitattu 29.8.2019].

Sharma, T. 2018. Understanding Redux + React in Easiest Way Part-1. WWW-dokumentti. Saatavissa: https://medium.com/tkssharma/understanding-redux-react-in-easiest-way-part-1-81f3209fc0e5 [viitattu 26.8.2019].

Shilman, M. 2017. The Storybook Story. WWW-dokumentti. Saatavissa: https://medium.com/storybookjs/the-storybook-story-dd3c1ab0d2ce [viitattu 27.8.2019].

Shilman, M. 2018. Storybook 4.0 is here! WWW-dokumentti. Saatavissa: https://medium.com/storybookjs/storybook-4-0-is-here-10b9857fc7de [viitattu 27.8.2019].

Stackshare. 2019. Who uses React? WWW-dokumentti. Saatavissa: https://stackshare.io/react [viitattu 27.6.2019].

Storybook. 2019a. TypeScript Config. WWW-dokumentti. Saatavissa: https://storybook.js.org/docs/basics/introduction/ [viitattu 27.8.2019].

Storybook. 2019b. Supercharge Storybook. WWW-dokumentti. Saatavissa: https://storybook.js.org/addons/ [viitattu 28.8.2019].

Storybook. 2019c. Introduction: React UI Testing. WWW-dokumentti. Saatavissa: https://storybook.js.org/docs/testing/react-ui-testing/ [viitattu 30.8.2019].

Storybook. 2019d. Structural Testing. WWW-dokumentti. Saatavissa: https://storybook.js.org/docs/testing/structural-testing/ [viitattu 30.8.2019].

Storybook. 2019e. Interaction Testing. WWW-dokumentti. Saatavissa: https://storybook.js.org/docs/testing/interaction-testing/ [viitattu 30.8.2019].

Storybook. 2019f. Automated Visual Testing. WWW-dokumentti. Saatavissa: https://storybook.js.org/docs/testing/automated-visual-testing/ [viitattu 30.8.2019].

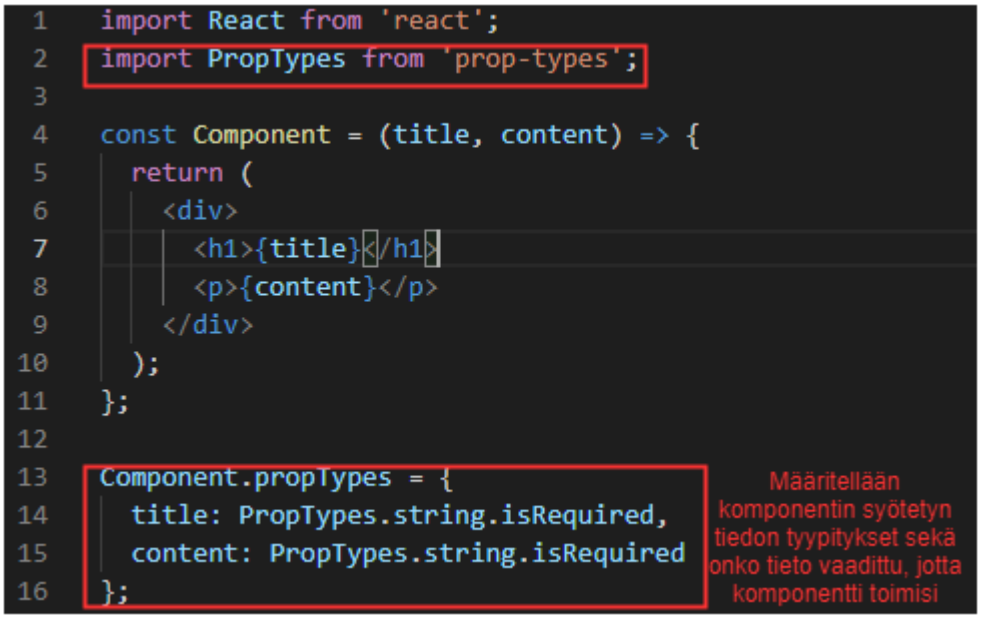
Storybook. 2019g. Manual Testing. WWW-dokumentti. Saatavissa: https://storybook.js.org/docs/testing/manual-testing/ [viitattu 30.8.2019].

The History of React and Flux with Dan Abramov. 2015. Three Devs and a Maybe. HAASTATTELU. Saatavissa: https://threedevsandamaybe.com/the-history-of-react-and-flux-with-dan-abramov/ [viitattu 26.8.2019].

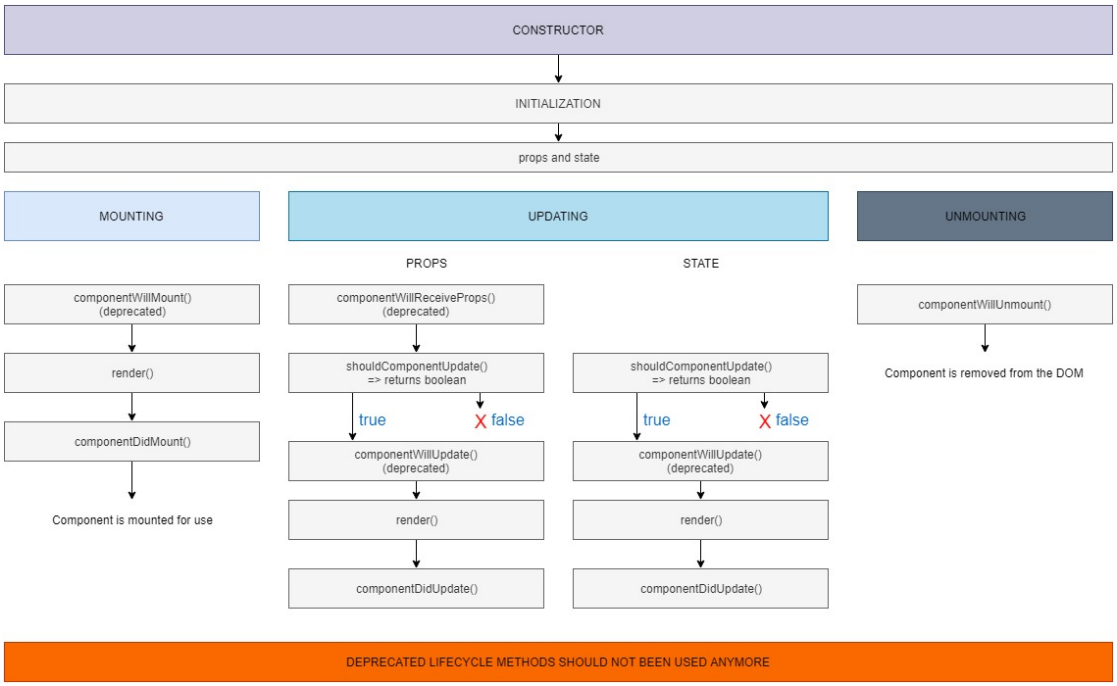
Webpack. 2019. Concepts. WWW-dokumentti. Saatavissa: https://webpack.js.org/concepts/ [viitattu 6.9.2019].

Whatley, W. 2018. React component patterns. WWW-dokumentti. Saatavissa: https://medium.com/teamsubchannel/react-component-patterns-e7fb75be7bb0 [viitattu 4.9.2019].

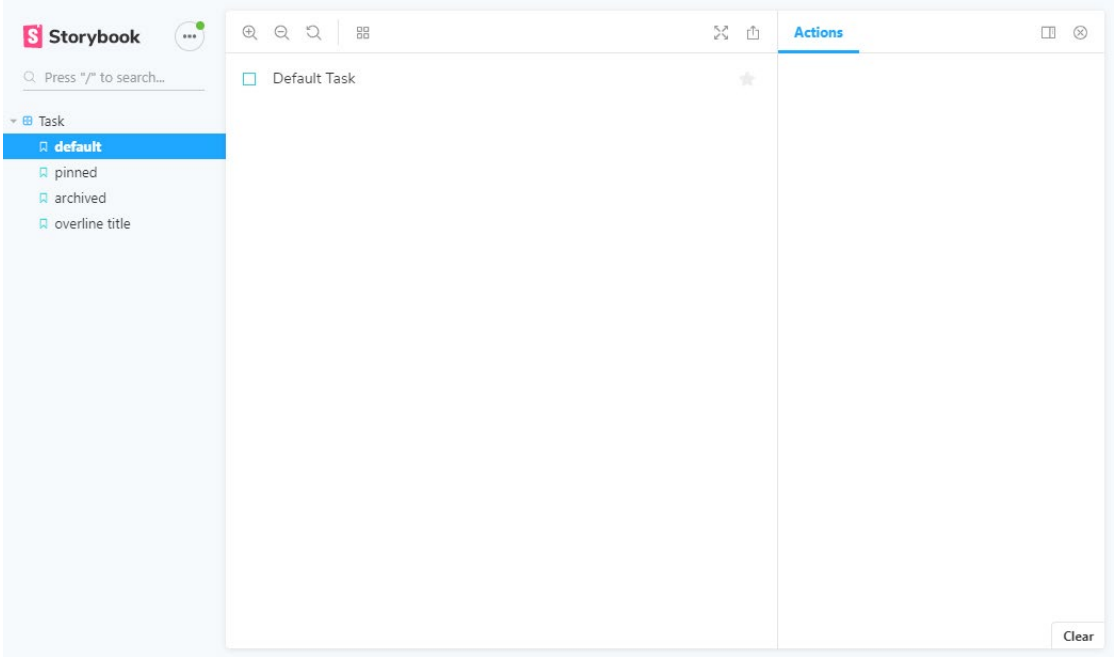
Komponentin ominaisuuksien tyypitys JavaScriptillä.



LIFECYCLE -metodit



Storybook -työskentelynäkymä



Storybook liitännäisten asennuskomennot

```
npm install @storybook/addon-info -D --save
npm install @types/storybook__addon-info -D --save
```
Addon info

```
npm install storybook-addon-react-docgen -D --save
npm install react-docgen-typescript-loader -D --save
npm install react-docgen-typescript-webpack-plugin -D --save
```
Addon Docgen

```
npm install addon-storysource/loader -D --save
npm install @storybook/addon-storysource -D --save
```
Addon Storysource

```
npm install @storybook/addon-knobs -D --save
npm install @types/storybook__addon-knobs -D --save
```
Addon Knobs

```
npm install @storybook/addon-notes -D --save
```
Addon Notes

```
npm install @storybook/addon-actions -D --save
```
Addon Actions

```
npm install storybook-addon-material-ui -D --save
```
Addon Material UI Theme

```
npm install @storybook/addon-storyshots-puppeteer -D --save
npm install @types/storybook__addon-storyshots-puppeteer -D --save
```
Addon Storyshots Puppeteer

```
npm install @storybook/addon-storyshots -D --save
npm install @types/storybook__addon-storyshots -D --save
```
Addon Storyshots

REDUX MOCK-malli

```
import { reducers, types } from 'ob2-shared';
import { combineReducers, createStore, ReducersMapObject } from 'redux';
import {
  getDefaultDevices,
  getDefaultDeviceStatusList,
  getDefaultDmConfigs,
} from '../__helpers__/mockHelpers';

interface ExtendedApplicationState extends Partial<reducers.ApplicationState> {}

/**
 * Creates mock store for storybook use
 * @param applicationState
 * @param state
 */
export const createMockStore = (
  applicationState: ReducersMapObject<ExtendedApplicationState, any>,
  state: any,
) => {
  const combinedReducers = combineReducers<ExtendedApplicationState>(
    applicationState,
  );

  return createStore(combinedReducers, state);
};

/**
 * Redux state mocks
 */
export const defaultContextState: reducers.ContextState = {
  primaryContext: {
    type: types.context.ContextType.CABINETS,
    context: types.context.CabinetContext.SINGLE_CABINET,
  },
  selectedItem: {
    deviceManagerKey: 'VBOX',
    type: types.context.SelectedItemType.CABINET,
  } as types.context.CabinetItem,
};

/**
 * Default provider mock data
 */
export const defaultDeviceInfoProvider: reducers.DeviceInfoState = {
  devices: getDefaultDevices(),
  deviceManagerConfigurations: getDefaultDmConfigs(),
};

export const defaultDeviceState: reducers.DeviceState = {
  deviceStates: getDefaultDeviceStatusList(),
};
```