Jouni Vanhanen

Verkkokäyttöliittymän ohjelmistosuunnittelu

Opinnäytetyö
Kevät 2021
SeAMK Tekniikka
Tietotekniikan tutkinto-ohjelma

SeAMK
SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: SeAMK Tekniikka

Tutkinto-ohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Tekijä: Vanhanen, Jouni Lauri Aleksi

Työn nimi: Verkkokäyttöliittymän ohjelmistosuunnittelu

Ohjaaja: Hilkka Niemelä

Vuosi: 2021 Sivumäärä: 75 Liitteiden lukumäärä: 0

Opinnäytetyön aiheena oli verkkosovelluksen ja sen käyttöliittymän suunnitteluun liittyvät tekniikat ja periaatteet. Aihetta pohjustettiin perehtymällä ohjelmistosuunnittelun suunnittelumalleihin ja arkkitehtuureihin. Arkkitehtuurien yhteydessä suunnittelun merkitystä konkretisoitiin esittelemällä ohjelmiston laatutekijät. Ohjelmiston yleisen rakenteen esittelyn jälkeen rajattiin tarkastelualue ohjelman esitystä kosketaviin tekijöihin. Esitysmalliesta kuvattiin MVC-, MVU- ja MVVM-mallit ja tutustuttiin Aurelia-verkkosovelluskehykseen, joka soveltaa MVVM-mallin periaatteita.

Työssä tutustuttiin myös verkkokäyttöliittymien toteuttamiseen tarkoitettuun Bootstrap-kirjastoon. Lopuksi esiteltiin työssä esitellyn teorian pohjalta toteutettua verkkokäyttöliittymää ja kerrottiin missä tuotettu verkkosovellus on nähtävissä ja kuinka sen kehitykseen voi osallistua.

Avainsanat: käyttöliittymäsuunnittelu, verkko-ohjelmointi, ohjelmistosuunnittelu, arkkitehtuuri

SEINÄJOKI UNIVERSITY OF APPLIED SCIENES

Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Software Engineering

Author: Vanhanen, Jouni Lauri Aleksi

Title of thesis: Software design of a web user interface

Supervisor(s): Hilkka Niemelä

Year: 2021 Number of pages: 75 Number of appendices: 0

The subject of the thesis was the software design of a web application and its web user interface. At first software- and architecture design patterns were introduced. The importance of software design was concretized by introducing software quality attributes in the chapter regarding software architecture. After introducing the general structure of a software system, the scope was narrowed down focusing only on the aspects related to the presentation layer. Aurelia framework, which applies the principles of the MVVM-pattern, was demonstrated after introducing it along with the presentation patterns. A library for implementing the user interface of a web application called Bootstrap was presented. And finally, a web user interface based on the theory presented in the thesis was demonstrated. It was also told where to find the project produced in the thesis and how to participate in the further development of the project.

Keywords: user interface design, web programming, software design, architecture

SISÄLLYS

Kuvioluettelo

Käytetyt termit ja lyhenteet

Abstraktio Yleistys jostain konkreettisesta tai teknisestä. Abstrahoinnin tarkoituksena on yksinkertaistaa poistamalla turhaa informaatiota.

Client-side Asiakas-palvelin-arkkitehtuuria noudattavissa web-sovelluksissa tarkoittaa operaatiota, joka suoritetaan loppukäyttäjän laitteella. Yleinen esimerkki on käyttöliittymän kuvantaminen.

Deklaratiivinen Imperatiivisen (ohjelmointiparadigman) vastakohta, jossa ohjelmalogiikalla kuvataan haluttua lopputulosta, eikä niinkään välivaiheita tuloksen saavuttamiseksi. Merkintäkielet, kuten HTML tai tietokannan kyselykielet kuten SQL, ovat esimerkkejä deklaratiivisista ohjelmointikielistä.

ECMAScript JavaScript standardi ja ohjelmointikieli takaamaan verkkosivujen yhteensopivuus eri selainten välillä.

Kontrolli Visuaalinen elementti käyttöliittymässä, jonka välityksellä käyttäjä voi vuorovaikuttaa järjestelmän kanssa. Esimerkiksi napit, syötekentät, linkit jne.

Metodi Olio-ohjelmoinnissa luokkaan assosioitu funktio, jota kutsutaan olion instanssin tai sen tyypin kautta. Metodien suorituskontekstista on myös implisiittinen pääsy omistavan luokan jäseniin, yleensä avainsanoilla kuten *this* tai *self*, jotka ovat käytännössä osoitin olioon itseensä.

Moduuli Ohjelmistokomponentti tai ohjelman osa, jossa on vähintään yksi rutiini.

Monoliitti Ohjelmisto joka koostuu vain yhdestä tieristä, eli sen loogiset ja ajonaikaiset komponentit muodostavat yhden itsenäisen kokonaisuuden. Monoliittisen sovelluksen kytkökset

	ovat vahvoja verrattuna modulaarisen sovelluksen löysiin kytköksiin.
Paradigma	Tutkijayhteisössä vallitsevien periaatteiden, uskomusten, arvostusten ja tieteellisten normien kokonaisuus.
POXO	Plain Old X Object, jossa *X* edustaa mielivaltaista ohjelmointikieltä. Tarkoittaa yksinkertaista oliota, jonka tyyppimääre sisältää pelkkää julkista dataa, eikä esimerkiksi toiminnallisuutta tai omaa sisäistä tilaa. Käytetään usein tietomallien yhteydessä. Ohjelmistoissa riippuvuussuhteet osoittavat yleensä yksinkertaisiin olioihin, mutta ei niistä ulospäin.
Property	Olio-ohjelmointikielissä luokan jäsen, joka on toiminnallisuudeltaan kentän ja metodin välimuoto. Käytännössä se on kenttä, jolla on metodit arvon lukemiseen ja asettamiseen ja jonka kutsun yhteyteen voidaan sijoittaa mielivaltaista logiikkaa, kuten validointia.
Responsiivisuus	Käyttöliittymien yhteydessä käyttöliittymän ulkoasun kyky mukautua käyttäjäystävälliseksi päätelaitteesta riippumatta. Esimerkiksi verkkosivu, jonka käyttökokemus säilyy yhtä hyvänä niin puhelimella kuin tietokoneella.
Sivuvaikutus	Ohjelmoinnissa sivuvaikutus kuvaa tilannetta, jossa suoritettava logiikka muuttaa oman näkyvyysalueensa ulkopuolella määritettyä tilaa.
Sovelluskehys	Alustaohjelma, kirjasto, tai kokoelmalogiikkaa, jota käytetään muiden ohjelmien tuottamisessa. Tarjoaa yleispätevän ratkaisun, tietylle ongelma-alueelle. Esimerkiksi JavaScript-ekosysteemissä yleinen kirjasto Moment.js, jonka ominaisuuksiin kuuluu mm. aikaleimojen käsittely, parsiminen, validointi, esittäminen jne.

Tagged union Summatyyppinen tietorakenne, joka mahdollistaa uuden tyypin määrittämisen joukosta muita tyyppejä. Hyödyllinen verrattuna lueteltuun tyyppiin (enum) esimerkiksi kielissä, joissa kääntäjä estää ohjelmaa kääntymästä, jos summatyyppiä käytetään huomioimatta kaikkia mahdollisia summatyypin tyyppitapauksia.

Tekninen velka Kuvaa helpon tai puutteellisen ratkaisun kustannuksia, jotka ilmenevät esim. kehityksen hidastumisena tai tulevaisuudessa vaadittavina korjaustöinä.

TypeScript Microsoftin kehittämä ja ylläpitämä JavaScriptin ylijoukko, joka tarjoaa käännösaikaisen tyyppitarkistuksen.

Viewport Sivun sisällön kuvantamiseen saatavilla oleva ruututila, eli selainikkunan koko vähennettynä työkalu- ja vierityspalkin ko'oilla.

1 JOHDANTO

Kun ohjelmisto toteutetaan oikein, sen tuottamiseen vaaditaan vain murto-osa työstä tai kustannuksista verrattuna järjestelmään, joka on syystä tai toisesta toteutettu suunnitteluun kunnolla panostamatta ja ilman harkittua arkkitehtuuria. Tällaiseen ohjelmistoon tehtävät muutokset osoittautuvat helposti monimutkaisiksi ja virhealttiiksi. (Martin 2017; Vernon 2016.)

Olio-ohjelmiston suunnittelu on haastavaa. On löydettävä olennaiset oliot ja rajattava ne luokiksi juuri oikealla tarkkuudella, määriteltävä luokkarajapinnat ja periytyvyyshierarkiat, sekä muodostettava merkittävät relaatiot näiden välille. Ratkaisun täytyy olla tarpeeksi kohdennettu ratkaistavaan ongelmaan, mutta tarpeeksi abstrakti taipuakseen myös uusiin vaatimuksiin tulevaisuudessa. On siis pyrittävä välttämään joutumasta suunnittelemaan uusiksi. (Johnson ym. 1994.)

Taitavat ohjelmoijat ja suunnittelijat ovat oppineet välttämään jokaisen ongelman ratkaisemista alusta alken. Sen sijaan he tietävät toistaa ratkaisuja, jotka ovat todistetusti toimineet heille tai muille jo aiemmin. He pyrkivät niin sanotusti välttämään keksimästä pyörää uudelleen. Tämän takia esimerkiksi olio-järjestelmissä on havaittavissa toistuvia luokkien malleja ja keskenään samoilla tavoin kommunikoivia olioita. Näitä toistuvia piirteitä kutsutaan suunnittelumalleiksi ja ne esittävät ratkaisuja tiettyihin ja rajattuihin ongelmiin suunnittelun tasolla. Suunnittelumallien avulla saadaan esimerkiksi tuotettua olio-ohjelmistoista joustavampia, helpommin omaksuttavampia ja ennalta arvattavampia. Ne siis auttavat ratkaisemaan yleisesti tunnettuja haasteita ja tarpeita ohjelmistokehityksessä. (Johnson ym. 1994.)

1.1 Tausta

Ohjelmistotekniikka ja erityisesti verkkosovellukset ovat hyvin nopeasti ja alati kehittyviä alueita. Jopa niin nopeita, että verkkosovelluksiin tarkoitetun ohjelmointikielen JavaScriptin ympärille on muodostunut termi JavaScript-väsymys, jolla viitataan tahtiin, jolla yleistä suosiota nauttivat verkkosovellustekniikat ja -työkalut vaihtuvat alalla sekä siitä koituvaan työhön ja rasitukseen.

Tästä syystä ohjelmoijalle on huomattava etu kyetä hahmottamaan ratkaistavat pulmat omaa työkaluaan korkeammalla tasolla. Missä ratkaisuun käytettävät työkalut saattavat vaihtua, pysyvät tarpeet ja periaatteet työkalujen takana hyvin pitkälti muuttumattomina.

Opinnäytetyö päätettiin toteuttaa teoreettisena tarkasteluna, sillä työnkuvan, ongelma-alueen ja käytettävien tekniikoiden vaihtuuvuus ohjelmistoja tuottavissa yrityksissä oli suuri. Laskentatavasta riippuen huomattiin ainakin viisi erilaista pinoa toteuttaa käyttöliittymiä muutaman vuoden sisällä.

1.2 Tavoite

Työn tavoitteena oli muodostaa alusta- ja työkaluriippumaton käsitys ohjelmiston ja erityisesti käyttöliittymän suunnittelemiseen liittyvistä yleisistä tekniikoista, toteutustavoista ja suunnitteluperiaatteista. Lisäksi tavoitteena oli soveltaa näitä tekniikoita verkkosovelluksen ja -käyttöliittymän toteutukseen tarkoitetuilla sovelluskehyksillä Aurelialla ja Bootstrapilla.

1.3 Rakenne

Luvussa 2 esitellään suunnittelumallit yleisesti ja tarkemmin olio-ohjelmoinnin kontekstissa. Luvussa 3 esitellään ohjelmistoarkkitehtuurin laatutekijät ja korostetaan siten suunnittelun merkitystä ohjelmoijalle ja ohjelmistotuotannolle. Luvussa 4 laajennetaan suunnittelumallien tarkastelualuetta koko ohjelmiston kattavaksi ja esitellään arkkitehtuurimallit. Luvussa 5 siirrytään ohjelmistoarkkitehtuurin kerrosmallin sisällä esityskerrokselle ja esitellään kolme rinnakkaista esitysmallia käyttöliittymien suunnitteluun.

Luvussa 6 tutustutaan Aurelia-sovelluskehykseen, joka on työkalu client-side-verkkosovellusten luomiseen ja jonka periaatteissa ammennetaan merkittävästi esitysmalleissa esiteltävää MVVM-mallia. Luvussa 7 tutustutaan verkkosivujen käyttöliittymien ulkoasun toteutukseen tarkoitettuun Bootstrapiin. Lisäksi esitellään kirjaston tarjoamia ratkaisuja käyttöliittymän asetteluun, mukautuvuuteen, tyylittelyyn ja muutamiin komponentteihin.

Luvussa 8 määritetään ja esitetään toteutettu verkkokäyttöliittymä, jonka tekemiseen on sovellettu opinnäytetyössä esiteltyä teoriaa ja tekniikoita erityisesti Aurelia- ja Bootstrap-sovelluskehyksiä. Luvussa kerrotaan myös miten ja mistä lopputuotetta pystyy tarkastelemaan ja kehittämään.

2 SUUNNITTELUMALLI

Suunnittelumallit eivät rajoitu ohjelmistokehitykseen vaan ne ovat yleispätevä ja abstrakti käsite, mutta tässä opinnäytetyössä niitä käsitellään ohjelmistotekniikan näkökulmasta ja tässä luvussa erityisesti olio-ohjelmointiparadigman kontekstissa.

Suunnittelumalli on yleinen uudelleen käytettävä ratkaisu jonkin käyttökohteen toistuviin ongelmiin tai tarpeisiin. Ongelma voi olla esimerkiksi, vaikka tarve saada vaihdettua tietyn logiikan sisällä käytettävää algoritmia ajon aikana, toistamatta algoritmeille yhteistä ohjelmalogiikkaa. Esimerkiksi tuotteen alennetun hinnan laskemiseen saatetaan tarvita erilaisia toteutuksia, mutta kaikkia eri toteutuksia on pystyttävä käyttämään yhden ja saman rajapinnan avulla. Ongelma on ratkaistavissa Gang of Fourin olio-ohjelmointisuunnittelumallikirjan katalogissa esitellyllä strategiamallilla (Johnson ym. 1994).

Suunnittelumallien avulla kehittäjä voi toistaa onnistuneita suunnitelmia, arkkitehtuureja ja toteutuksia. Suunnittelumalleiksi yksinkertaistetut toistuvat tekniikat ovat lähestyttävämpiä uusien järjestelmien kehittäjille. Suunnittelumallit auttavat tekemään päätöksiä, jotka parantavat järjestelmän (uudelleen)käytettävyyttä. Suunnittelumallit saattavat auttaa olemassa olevan järjestelmän dokumentoinnissa ja ylläpidossa luomalla yksiselitteiset määreet luokkien ja ilmentymien vuorovaikutuksille. Suunnittelumallit auttavat myös kuvaamalla järjestelmän osien tarkoitusta. (Johnson ym. 1994.)

Suunnittelumallilla on neljä olennaista piirrettä:

Nimi, jota voidaan käyttää kuvaamaan suunnittelun ongelmaa, sen ratkaisuja ja seurauksia sanalla tai kahdella. Suunnittelumallin nimeäminen kasvattaa suunnittelusanastoa ja mahdollistaa suunnittelun korkeammalla abstraktiotasolla. Nimeämällä konseptit on niistä mahdollista viestiä sujuvammin esimerkiksi kollegoille, dokumentoinnissa tai itsekseen. Nimet siis helpottavat suunnittelua ja siitä viestimistä. (Johnson ym. 1994; Raman, Subramanian & Raj 2017.)

Ongelma kertoo milloin soveltaa suunnittelumallia. Se selittää ongelman kontekstin. Se saattaa kuvailla erityisiä haasteita, kuten kuinka kuvata algoritmeja olioina. Se

voi kuvailla luokka- tai oliorakenteita, jotka enteilevät kankeaa suunnittelua. Joskus ongelma voi sisältää listan ehtoja, joiden on täytyttävä ennen kuin on järkevää soveltaa mallia. (Johnson ym. 1994; Raman ym. 2017.)

Ratkaisu kuvailee ominaisuuksia, jotka luovat suunnittelumallin, niiden relaatioita, vastuita ja yhteistyötä. Ratkaisu ei kuvaile tiettyä konkreettista toteutusta, koska malli on kuin sapluuna, jota voidaan soveltaa moneen eri tilanteeseen. Malli tarjoaa abstraktin selityksen suunnittelupulmasta ja kuinka se on ratkaistavissa tietyin osatekijöin, kuten esimerkiksi luokin ja olioin. (Johnson ym. 1994; Raman ym. 2017.)

Seuraukset kuvaavat mallin soveltamisen mahdolliset hyödyt sekä haitat. Vaikka seuraukset jäävät usein vähälle äänelle suunnitteluratkaisua kuvailtaessa, ovat ne kriittisiä vaihtoehtoisten toteutusten arvioinnin kannalta ja mallin soveltamisen vaikutusten ymmärtämiselle. (Johnson ym. 1994; Raman ym. 2017.)

Suunnittelumalli nimeää, abstrahoi ja erittelee yleisen suunnitelurakenteen avaintekijät, jotka tekevät siitä hyödyllisen uudelleenkäytettävän toteutuksen luonnissa. Jokainen suunnittelumalli keskittyy tiettyyn yleistettävissä olevaan tarpeeseen. Olioohjelmoinnissa suunnittelumalli esimerkiksi erittelee vuorovaikutuksessa olevat luokat ja ilmentymät, niiden roolit ja toiminnan yhdessä, sekä vastuun jaon niiden välillä. (Johnson ym. 1994.)

3 OHJELMISTOARKKITEHTUURI

Järjestelmän ohjelmistoarkkitehtuuri on joukko merkittäviä päätöksiä siitä, kuinka ohjelmisto rakennetaan niin, että se palvelee laatutekijöitä ja toteuttaa siltä vaaditut ominaisuudet ja toiminnallisuudet (Keeling 2017). Arkkitehtuurin päämääränä on minimoida vaaditun järjestelmän tuottamiseen ja ylläpitämiseen vaadittavat inhimilliset voimavarat (Martin 2017).

Suunnittelun aikainen päätös voi olla merkittävä monestakin syystä. Se voi tarkoittaa valintaa, jota ei voida peruuttaa. Se voi vaikuttaa hankkeen laatutekijöihin, aikatauluun tai kuluihin. Päätös voi olla myös merkittävä, jos se koskettaa monia henkilöitä tai aiheuttaa muutoksia muihin järjestelmiin. (Keeling 2017; Martin 2017.) Oli tapaus mikä hyvänsä, merkittävät päätökset ovat kalliita muuttaa jälkeenpäin.

Ohjelmiston laatutekijät ovat järjestelmän konkreettisia ja mitattavia ominaisuuksia joihin lukeutuvat mm:

Kehitys. Hankalasti kehitettävissä olevan järjestelmän elinkaari tuskin tulee olemaan kovin pitkä tai kivuton, joten järjestelmän arkkitehtuurin tulisi tehdä sen kehittämisestä helppoa kehittäjilleen. (Martin 2017.)

Julkaiseminen. Ohjelmiston toimivuuden ja kannattavuuden edellytyksenä on sen helppo jakelu. Mitä suuremmat jakelukustannukset ovat, sitä hyödyttömämpi järjestelmä on. Täten yksi ohjelmistoarkkitehtuurin päämääristä tulisi olla helposti ja yhdellä toimenpiteellä julkaistavan järjestelmän tuottaminen. (Martin 2017.)

Suoritus. Arkkitehtuurin vaikutus järjestelmän suorituskykyyn tapaa olla vähemmän dramaattinen, kuin sen vaikutus kehitykseen, jakeluun tai ylläpidettävyyteen. Miltei kaikki suorituskyvylliset haasteet ovat ratkaistavissa hankkimalla järjestelmään tehokkaamman laitteiston aiheuttamatta juurikaan arkkitehtuurillisia toimenpiteitä. (Martin 2017.)

Ylläpito. Kaikista ohjelmistojärjestelmän laatutekijöistä kallein on ylläpito. Alati tyrehtymätön uusien toiminnallisuuksien virta, väistämättömät virheet ja niistä selviytyminen kuluttavat valtavan määrän inhimillisiä voimavaroja. (Martin 2017.)

Ylläpidon kustannuksen koituvat ensikädessä loputtomasta ohjelmistoon perehtymisestä ja ohjelmistokehityksen riskeistä. Loputtoman perehtymisen kustannukset ovat ne, jotka syntyvät olemassa olevan koodin tulkkaamisesta, etsien parasta paikkaa ja tapaa toteuttaa uusi toiminallisuus tai suorittaa virheen korjaus. Riskit puolestaan tarkoittavat ohjelmiston muokkaamisesta mahdollisesti koituvia kustannuksia uusien tahattomien virheiden muodossa. (Martin 2017.)

Huolellisesti ja läpikotaisin suunniteltu arkkitehtuuri vähentää näitä kustannuksia tuntuvasti. Rajaamalla järjestelmä moduuleihin ja yhdistämällä ne vakailla rajapinnoilla, on mahdollista keventää ohjelmiston sisäistämisen taakkaa, helpottaa tulevien toiminnallisuuksien toteuttamista ja huomattavasti madaltaa riskiä ohjelmiston tahattomalle rikkoontumiselle. (Martin 2017.)

Vaihtoehtojen säilyttäminen avoimena. Ohjelmiston arvo muodostuu kahdesta tekijästä, sen toimintojen arvosta ja ohjelmiston rakenteen arvosta (Martin 2017). Arvoista merkittävämpi on rakenteellinen arvo. Tämän käsitteen ymmärtämiseen on ensin selitettävä englanninkielistä termistöä.

Englanniksi laitteisto muodostuu osista: ***soft****ware*, ***firm****ware*, ja ***hard****ware*. Sen muodostavissa sanoissa esiintyy selvästi yhteinen etuliite, joka kuvaa kunkin osan mukautuvuutta:

- *Software* eli laitteella ajettava ohjelmisto kuvataan ”pehmeäksi”. Tämän voidaan ajatella viestivän, että se on tarkoitettu muovattavaksi tarpeen niin vaatiessa digitaalisesti ja kajoamatta laitteiston fyysisiin ominaisuuksiin. (Martin 2017.)
- *Firmware* eli laiteohjelmisto kuvataan ”jämäkäksi”, jonka voidaan ajatella johtuvan siitä, että laiteohjelmisto on laitteen perustoimintoja ohjaava osa, joka ei alun perin ollut muutettavissa, ja on vahvasti liitoksissa fyysisiin komponentteihin (Martin 2017).
- *Hardware* eli laitteiston fyysiset komponentit kuvataan ”kiinteäksi”. Laitteen toimintaan vaikuttaminen tällä tasolla tarkoittaa käytännössä kiinteiden osien vaihtamista. (Martin 2017.)

Alun perin ohjelmistot kehitettiin tarpeeseen saada muutettua laitteiden toiminnallisuuksia helposti ja nopeasti ilman fyysisiä toimenpiteitä. Se, mikä mahdollistaa ohjelmiston mukautuvuuden ja ns. pehmeyden, on ohjelmiston toteuttaminen sitomatta sitä merkityksettömiin yksityiskohtiin pitäen kaikki mahdolliset vaihtoehdot avoinna. Kaikki ohjelmistojärjestelmät voidaan purkaa kahteen päätekijään: toimintaperiaatteeseen (policy) ja yksityiskohtiin. (Martin 2017.)

Toimintaperiaatteet kattavat kaikki liiketoimintalogiikan säännöt ja proseduurit. Järjestelmän todelliset arvot muodostuvat siis toimintaperiaatteista. (Martin 2017).

Yksityiskohdat ovat niitä asioita joita ihmiset, toiset järjestelmät ja ohjelmoijat edellyttävät viestiäkseen toimintaperiaatteiden kanssa. Yksityiskohtia ovat mm. tietokannat, palvelimet, sovelluskehykset, viestintäprotokollat ja niin edespäin. Esimerkiksi tiedon pysyvyyden kannalta järjestelmä riippuu jostain konkreettisesta tietokantamoottorista, mutta tietokantamoottorivalinta ei saa vaikuttaa järjestelmän liiketoimintalogiikkaan. (Martin 2017.)

Arkkitehdin päämäärä on luoda järjestelmälle muoto, joka tunnustaa toimintaperiaatteet tärkeimmäksi ominaisuudekseen tehden samalla järjestelmään liittyvistä yksityiskohdista merkityksettömiä toimintatavoille. Tämä mahdollistaa olemaan sitoutumatta yksityiskohtiin ennen kuin niitä on viimein tarve toteuttaa. (Martin 2017.)

4 ARKKITEHTUURIMALLI

Gang of Fourin -suunnittelumallit opastavat, kuinka rakentaa olio-ohjelmia uudelleenkäytettävyyden ja ylläpidettävyyden kannalta. Arkkitehtuurimallit puolestaan usein käsittelevät lukuisia komponentteja ohjelmistojärjestelmässä ja tarjoavat ratkaisuja laatutekijälliseltä kannalta. Arkkitehtuurimallissa laatutekijöiden ja abstraktiotasojen tarkastelun ulottuvuus on laajempi. Käytännössä kuitenkaan suunnittelumallien erotteleminen arkkitehtuurimalleista ei ole niin tärkeää, sillä raja niiden välillä on vaihteleva ja riippuu sovelluskohteesta ja tekijästä. (Richards 2015.)

Arkkitehtuurimallin tärkein päämäärä on ongelmien eriyttämisen periaate. Ongelmien eriyttäminen voidaan saavuttaa esimerkiksi jakamalla ohjelmisto kerroksiin. Jokaisessa arkkitehtuurimallissa on vähintään yksi kerros liiketoimintalogiikalle ja toinen rajapinnoille. (Martin 2012.)

Arkkitehtuurimallit pyrkivät tuottamaan järjestelmiä, jotka ovat:

Riippumattomia sovelluskehyksistä. Arkkitehtuurin tulee olla riippumaton ulkopuolisista kirjastoista. Tämä mahdollistaa sovelluskehysten käytön työkaluina sen sijaan, että ne asettaisivat rajoitteita tai ohjaisivat lopputuotteen kehitystä. (Martin 2012.)

Testattavissa. Liiketoimintalogiikka ja säännöt ovat testattavissa ilman käyttöliittymää, tietokantaa, web-palvelinta tai muuta ulkoista rajoittavaa tekijää. (Martin 2012.)

Riippumattomia käyttöliittymästä. Käyttöliittymää voidaan muuttaa ilman, että se aiheuttaa muutoksia muualle järjestelmään. Web-käyttöliittymän on oltava korvattavissa esimerkiksi komentoliittymällä tämän vaikuttamatta liiketoimintalogiikkaan. (Martin 2012.)

Tietokantariippumattomia. Tietokantamoottori on oltava vaihdettavissa, vaikka SQL Serveristä Redisiin tai Google Cloud Firestoresta Oracleen ja niin edespäin. Luonnollisesti myöskään liiketoimintalogiikka ei saa olla sidoksissa tietokantoihin. (Martin 2012.)

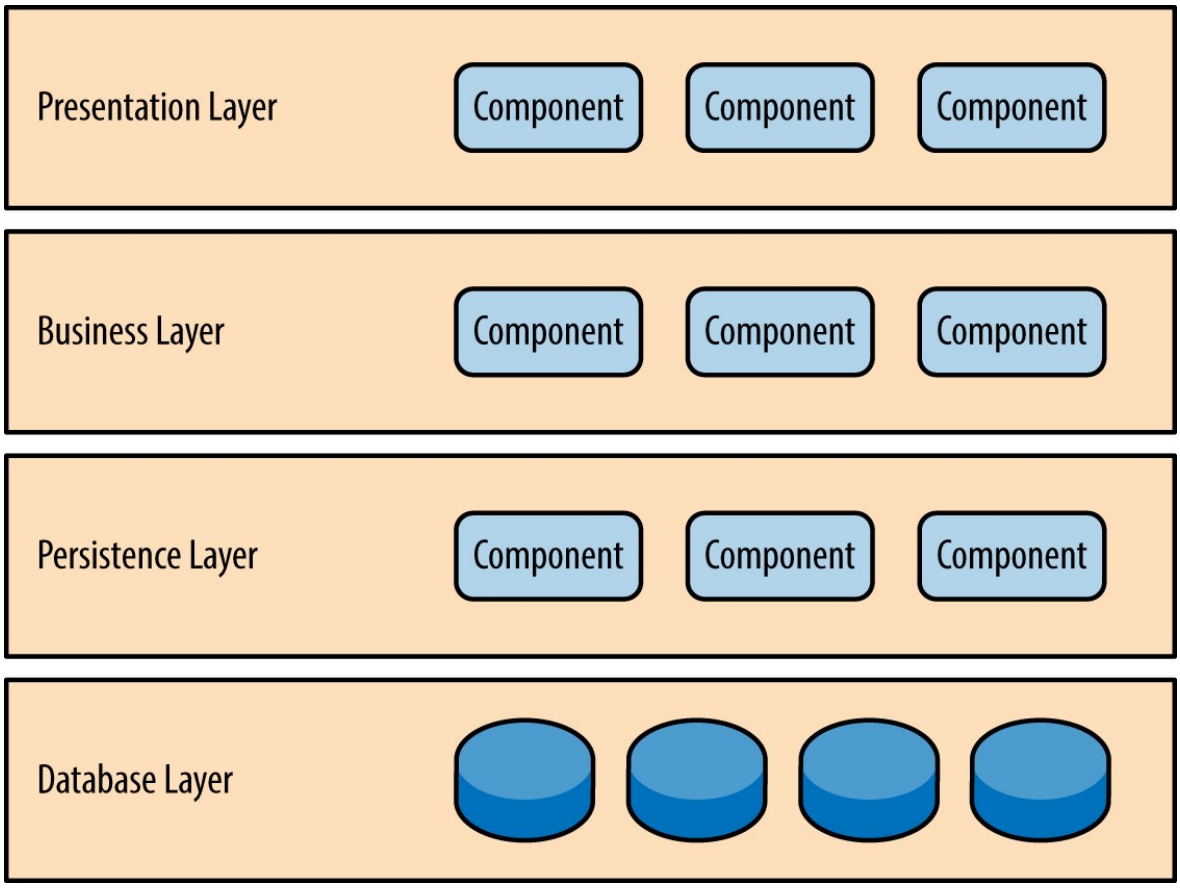
4.1 Kerrosarkkitehtuuri

Jakamalla sovellus selkeisiin erillisiin vastuualueisiin saadaan pyyntöjen kulkua sovelluksessa selkeytettyä. Kerrosarkkitehtuurissa (layered architecture) sovellus on jaettu itsenäisiin kerroksiin, jotka ovat vuorovaikutuksessa toistensa kanssa. (Agile Education Research 2019.)

Kerrosarkkitehtuurissa riippuvuussuhde kulkee ylhäältä alaspäin. Ylempänä olevat kerrokset hyödyntävät alempana olevia kerroksia, mutta eivät tiedä itseään korkeammalle sijoittuvista kerroksista, eivätkä siten myöskään käytä niitä. Puhtaassa kerrosarkkitehtuurissa kutsut eivät myöskään ohita kerroksia kulkiessaan, vaan kulkevat aina lähimmän kautta. (Agile Education Research 2019.)

Kerrosarkkitehtuuri ei suoraa määritä, montako tai millaisia kerroksia mallissa on oltava. Useimmiten kerrosarkkitehtuuri kuitenkin koostuu seuraavista neljästä kerroksesta: **esitys-, liiketoiminta-, pysyvyys- ja tietokantakerroksesta**. (Agile Education Research 2019.)

Kerrokset ovat havainnollisetttu kuviossa 1.



Kuvio 1. Kerrosarkkitehtuuri (Richards 2015).

Joissain tapauksissa liiketoiminta- ja pysyvyyskerros on yhdistetty yhdeksi kerrokseksi, etenkin silloin kun säilyvyyden logiikka on upotettuna bisnes-kerroksen komponentteihin. Muodostetut kerrokset riippuvat siis toteutettavan sovelluksen monimutkaisuudesta ja vaatimuksista. (Richards 2015.)

4.1.1 Esityskerros

Esityskerroksen tehtävänä on mm. tiedon visualisointi, palautteen viestiminen käyttäjälle, käyttäjän syötteiden vastaanottaminen ja vastaanotettujen syötteiden välittäminen käsiteltäväksi eteenpäin, useimmiten liiketoimintalogiikkakerrokselle. (Vice & Siddiqi 2012).

Esityksen eristäminen omaan kerrokseensa tarjoaa mahdollisuuden käyttöliittymän vaihtamiseen ilman, että alempien tasojen ohjelmakoodia joudutaan monistamaan tai muokkaamaan (Vice & Siddiqi 2012).

4.1.2 Liiketoimintalogiikkakerros

Liiketoimintalogiikkakerros tai applikaatiokerros on taso, jolla järjestelmän ydintoiminnallisuudet ovat määritelty. Tämän kerroksen logiikkaa kutsutaan bisnes- tai domainlogiikaksi ja sillä prosessoidaan tietoyhteyskerroksen (data access layer, DAL) kautta haettu data. (Vice & Siddiqi 2012.)

Liiketoimintalogiikan eristäminen omaan kerrokseensa tekee sovelluksesta skaalattavan, sillä kerros voidaan sijoittaa laitteistolle tai palvelimelle, joka on erillään muista kerroksista. Sovellusta on myös helpompi laajentaa tai muuttaa, sillä se on riippuvainen vain tietoon pääsystä, ei sen esittämisestä tai säilömisestä. Käyttöliittymät ja tietokantamoottorit ovat sen kannalta vaihdettavissa. (Vice & Siddiqi 2012.)

4.1.3 Tiedon pysyvyyskerros

Pysyvyyskerros on vastuussa tiedon hausta ja talletuksesta tietovarastoon kuten tietokantaan, palveluun tai XML-tiedostoon. Pysyvyyskerros mahdollistaa tiedon varastoinnin toteutuksen muuttamisen vaikuttamatta ylempien kerrosten toimintaan. (Vice & Siddiqi 2012.)

Jokaisella kerroksella kerrosarkkitehtuurissa on erityinen ja merkittävä rooli sekä vastuu sovelluksen rakenteesta ja toiminnasta. Jokainen kerros muodostaa abstraktion liiketoiminnan edellyttämän pyynnön täyttämiseen tarvittavalle työlle. (Richards 2015.)

Esityskerrokselle tämä tarkoittaa, että sen ei tarvitse huolehtia tai tietää kuinka hankkia asiakkaasta tietoa. Sen tarvitsee ainoastaan esittää tieto ruudulla halutunlaisessa formaatissa. (Richards 2015.)

Samaan tapaan liiketoimintalogiikkakerroksen ei tarvitse kantaa huolta siitä, kuinka muotoilla asiakkaan data näytettäväksi ruudulla tai edes tietää mistä data on peräisin. Sen tarvitsee ainoastaan pyytää data pysyvyyskerrokselta, prosessoida saatu tieto ja välittää tieto eteenpäin esittämiskerrokselle. (Richards 2015.)

4.2 Ajonaikainen kerrosarkkitehtuuri

Kerrosmallista on olemassa myös variantti, jossa kerroksia kutsutaan *Layereiden* sijaan *Tiereiksi*. Edellä mainitut ovat hyvin samankaltaisia, mutta tiered-mallissa ajonaikaiset rakenteet järjestetään loogisiin ryhmiin suunnittelunaikaisten rakenteiden sijaan. Nämä ajonaikaiset loogiset ryhmät saattavat olla allokoituja tietynlaisille fyysisille komponenteille, kuten palvelimille tai pilvialustoille. (Keeling 2017.)

Järjestelmissä, joissa komponentit sijaitsevat fyysisesti eri alustoilla tai laitteistoilla, hyödytään ajattelemalla kerroksia tiereinä ja suunnittelemalla mitkä komponentit sijaitsevat milläkin tierillä (Keeling 2017).

5 ESITYSMALLIT

Arkkitehtuurin yhteydessä esitelty kerrosarkkitehtuuri mallinsi läpileikkauksen yhdestä monoliittisesta järjestelmästä. Tässä luvussa keskitytään esityskerrokseen.

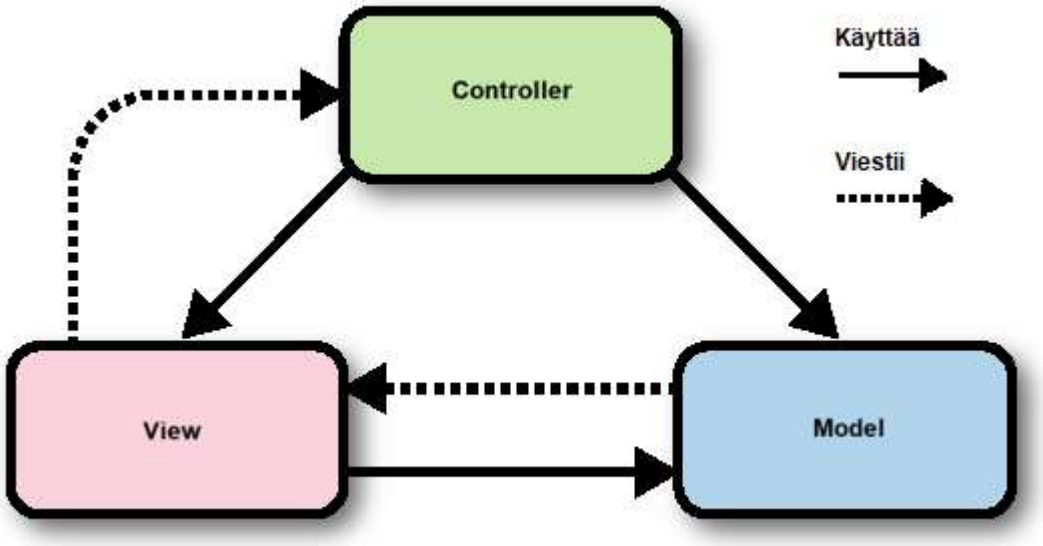
Luvussa ei ole tarkoitus käsitellä tapahtuuko, käyttöliittymän renderöinti asiakkaan vai palvelimen päässä. Luvussa ei käsitellä myöskään, kuinka käyttöliittymä kommunikoi palvelimen kanssa, mutta kaikkia tekniikoita yhdistää, että viimekädessä ne käyttävät service-luokkia kutsuakseen applikaatiokerrosta.

5.1 MVC

MVC-mallissa keskitytään jakamaan esittämiseen liittyvä logiikka kolmeen eri osaan (kuvio 2):

- **Tietomalleihin** (model), jotka antavat sovelluksen tilalle muodon ja ylläpitävät sitä.
- **Näkymiin** (view), joilla käyttäjälle visualisoidaan sovelluksen tila.
- **Käsittelijöihin** (controller), joiden vastuulla on käyttäjävuorovaikutuksen prosessointi. Ne toimivat myös ikään kuin siltoina tietomallien ja näkymien välillä. (Ingeno 2018; Warin 2015; Agile Education Research 2019.)

Mallin tavoitteena on käyttöliittymän ja sovelluslogiikan eriyttäminen toisistaan, ettei ne jakaisi toisilleen tärkeätä logiikkaa. Tietomallin on tarkoitus olla omavarainen ja käyttöliittymästä autuaan tietämätön. Tämä mahdollistaa saman tiedon näyttämisen erilaisissa näkymissä. Näkymä käyttäytyy siis vain tapana hahmottaa tieto eri muodossa. (Warin 2015; Agile Education Research 2019.)



Kuvio 2. Model-View-Controller yleiskuva (Hall 2011).

Noudattamalla MVC-mallia voidaan esityskerroksen osia työstää rinnakkain. Myös ohjelmalogiikka yksinkertaistuu, sillä sen esitysmallin mukainen toiminta on ennakoitavissa ilman tutustumista sovelluskohtaisiin yksityiskohtiin. (Warin 2015; Agile Education Research 2019.)

Model-View-Controller-malli tuli laajasti tunnetuksi noustessaan esiin Smalltalk-kielen yhteydessä ja päädyttyään Ruby on Rails -sovelluskehykseen (Warin 2015).

5.1.1 Model

Model eli **tietomalli** vastaa sovelluksen datan muodosta ja tilasta. Sen vastuisiin kuuluu tietovarastoon, kuten tietokantaan, menevän ja sieltä tulevan datan väliprosessointi. Tietomalli on riippumaton näkymistä ja käsittelijöistä, jolloin sen kehitys ja testaus on mahdollista itsenäisesti erillään muusta ohjelmasta. Tietomallit vastaanottavat käskyjä käsittelijöiltä, joiden kautta sovelluksen tilaa ohjataan. (Ingeno 2018; Warin 2015; Agile Education Research 2019.)

MVC-mallissa tietomalli saattaa olla joko aktiivinen tai passiivinen. Aktiivinen tietomalli ilmoittaa omatoimisesti näkymälle muutoksista tilaansa, kun taas muutokset passiiviseen tietomalliin eivät edellytä siltä toimenpiteitä. (Ingeno 2018.)

5.1.2 View

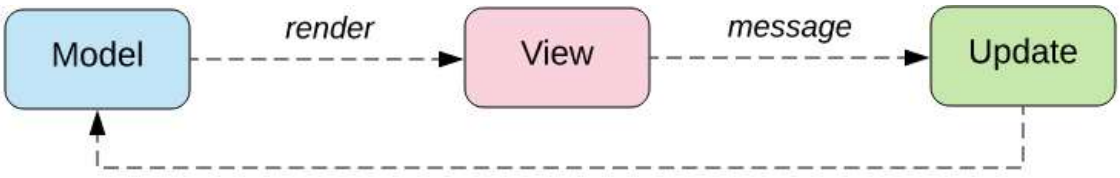
View eli **näkymä** vastaa sovelluksen ulkoasusta ja esittämisestä. Se on sovelluksen osa, joka on näkyvissä loppukäyttäjälle. Käsittelijän käskyttämänä näkymä esittää sovelluksen tilan halutulla ulkoasulla ja muodostaa rajapinnan käyttäjälle sovelluksen tilaan vaikuttamiseen. Tietomallin ollessa aktiivinen näkymä saattaa kuunnella tietomallin muutosviestejä ja päivittää esitystään niiden mukaan. Näkymä välittää tiedon käsittelijälle, kun käyttäjä on vuorovaikutuksessa näkymän kanssa esimerkiksi antamalla syötteitä. (Ingeno 2018; Warin 2015; Agile Education Research 2019.)

5.1.3 Controller

Controller eli **käsittelijä** toimii välikätenä tietomallin ja näkymän välillä. Käsittelijä suorittaa pyyntöjä, joita syntyy esimerkiksi käyttäjän navigoidessa verkkosovelluksen sivuilla. Käsittelijä selvittää sovelluslogiikalla tarvittavan näkymän ja välittää sille tarvittavat tiedot käyttöliittymän piirtämistä varten. Käsittelijät päivittävät tietomallia käyttäjän toimien perusteella. (Ingeno 2018; Warin 2015; Agile Education Research 2019.)

5.2 MVU

MVU eli Model-View-Update on esitysmalli, joka kulkee myös nimellä Elm-arkkitehtuuri. Tyypillisesti Elm-ohjelmat koostuvat **näkymäfunktiosta,** joka renderöi mallin HTML-kieleksi, **päivitysfunktiosta,** joka käsittelee kaikki malliin kohdistuvat muutokset ja **tietomallista**, joka kuvaa ja ylläpitää sovelluksen tilaa. (Imsirovic 2018; Fairbank 2019; Griffith 2017; Poudel 2018; Elm-lang 2020.)



Kuvio 3. Model-View-Update yleiskuva (Poudel 2018).

5.2.1 Model

Elm-arkkitehtuurissa tietomallin tyyppi on vapaasti valittavissa ja se voidaan julistaa esimerkiksi kokonaisluvuuksi tai merkkijonoksi, mutta tyypillisesti tietomallit ovat *Record*-tyyppiä (Fairbank 2019).

Tietue (record) on samankaltainen kuin POJO (Plain Old JavaScript Object). Se nitoo toisistaan riippuvat kentät yhteen avain-arvo-pareiksi. Elm-kehittäjät usein kutsuvat syötteitä (entry) tietueissa kentiksi (field) (Fairbank 2019).

Tietueissa merkittävä ero verrattuna POJO-objekteihin on se, että tietueet ovat muuttumattomia, mikä tunnetaan funktionaalisen paradigman ja kielten tunnusmerkkinä. Muuttumaton tietotyyppi ei voi ns. muuttua paikallaan (in-place), eli alustetun tietueen kenttiä tai niiden arvoja ei voi muuttaa, eikä siihen voi enää lisätä uusia kenttiä. Tietue ei myöskään salli määrittelemättömiä tai mitättömiä kenttiä kuten *undefined* tai *null*. (Fairbank 2019; Elm-lang 2020.)

Muuttumattoman tietomallin päivitys tapahtuu korvaamalla se uudella ilmentymällä, joka alustetaan muutetulla tilalla. Elmissä tietomalli päivittyy aina kun se on osana suoritettavaa funktiota. (Imsirovic 2018; Elm-lang 2020.)

Mitä Record-tietotyyppiin tulee, muuttumattomuus on vähintäänkin nosteessa, sillä perinteiset olio-ohjelmointikieletkin, kuten Java ja C# aikovat toteuttaa ja omaksua kyseisen ominaisuuden Java SE versiosta 14 ja C# puolestaan versiosta 9.0 alkaen. (Oracle. 2020; Microsoft 2020.)

5.2.2 View

Myös Elm-arkkitehtuurissa näkymää ajatellaan tapana visualisoida tietomalli ja rajapintana suorittaa sille ohjelman käyttötapausten mukaisia toimenpiteitä. Elm-arkkitehtuurin kohdalla poikkeavaa on kuitenkin se, että siinä näkymä on funktio. (Imsirovic 2018; Poudel 2018.)

Näkymäfunktio hyväksyy rajanpintansa mukaiset parametrit ja pyrkii tuottamaan viestin, joka sisältää näkymän esitysasun HTML-merkintäkielenä. Tuotettu viesti päätyy viimekädessä selaimen renderöintimoottorille kuvannettavaksi. (Imsirovic 2018; Poudel 2018; Elm-lang 2020).

```
view : Model -> Html msg
view model =
  div []
    [ button [] [ text "-" ]
    , text (String.fromInt model)
    , button [] [ text "+" ]
    ]
```

Kuvio 4. Näkymäfunktio Elmillä.

Kuviossa 4 näkymäfunktion sisällä tapahtuu seuraavaa:

Funktioiden ensimmäinen argumentti on lista attribuutteja. Toinen argumentti edustaa listaa sisäkkäisistä elementeistä. Div-funktio vastaanottaa tyhjän listan attribuutteja ja listan sisäkkäisistä elementeistään: kahdesta napista ja tekstistä. Div-funktio tuottaa `<div>`-elementin ja button-funktio `<button>`-elementin. Text-funktio ei tuota HTML-elementtiä, vaan se muuntaa mallin kokonaisluvun merkkijonoksi.

Kuviossa 5 nähdään selaimen dokumenttirakenteeseen lopulta muodostunut HTML-esitysasu.

```
<div>
 <button> + </button>
  String representation of our model
 <button> - </button>
</div>
```

Kuvio 5. Näkymäfunktion tuottama dokumenttirakenne.

`div`-, `button`- ja `text`-funktioiden määrittelyt sijaitsevat Html-moduulissa, joka kuuluu *elm/html*-pakettiin. Moduuli tarjoaa täyden pääsyn HTML-elementteihin Elm-funktioiden kautta. Koska Elmissä näkymän muodostava ohjelmalogiikka on myös kirjoitettu Elmillä, voidaan näkymäkoodiin soveltaa kaikkia kielen tarjoamia ominaisuuksia ja hyötyjä. Näkymiä voidaan esimerkiksi automaatiotestata samoilla työkaluilla, kuin kaikkia muitakin Elm-ohjelmiston osia. Myös Elm-kääntäjä on tukena näkymäkoodia tuotettaessa. (Poudel 2018.)

Näkymäfunktio ei vastaa HTML:n kuvittamisesta ruudulle. Funktio yksinkertaisuudessaan vastaanottaa mallin ja palauttaa palan HTML:ää. Se on puhdas funktio, joka tuottaa aina saman lopputuloksen samoille lähtöarvoille. Kuvittaakseen HTML:n Elm käyttää toista pakettia nimeltä *elm/virtual-dom.* (Poudel 2018.)

5.2.3 Update

Elmissä päivitysfunktio kuvaa kuinka tietomalli muuttuu ajan myötä. Päivitysfunktio vastaanottaa viestin, päättää kuinka päivittää sovelluksen tila, eli tietomallin viestin perusteella. Päivityksen ajon jälkeen uusi tietomalli on luotu ja se renderöi uuden näkymän. Käyttäjä saattaa sitten olla vuorovaikutuksessa luodun näkymän kanssa, josta syntyy uusi viesti ja MVU-sykli jatkuu. (Imsirovic 2018; Elm-lang 2020.)

```
type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
 case msg of
  Increment -> model + 1
  Decrement -> model - 1
```

Kuvio 6. Päivitysfunktio Elmissä.

Kuvion 6 esimerkissä nähdään päivitysfunktio, jonka rajapinta ottaa vastaan viestin ja tietomallin. Viestin tyyppi on määritelty korotus- ja vähennysoperaatioiden discriminated unionina. Vastaanotettaessa `Increment`-viesti palautetaan parametrina ollut tietomalli korotettuna yhdellä. `Decrement`-viesti toimii samoin, mutta suorittaa vähennysoperaation.

Merkittävää päivitysfunktiossa, on se, että tietomalli on funktion paluuarvona ja välittyy suoraan näkymälle, joka vuorostaan päättää, kuinka päivittynyt tila tulisi esittää ruudulla. (Elm-lang 2020.)

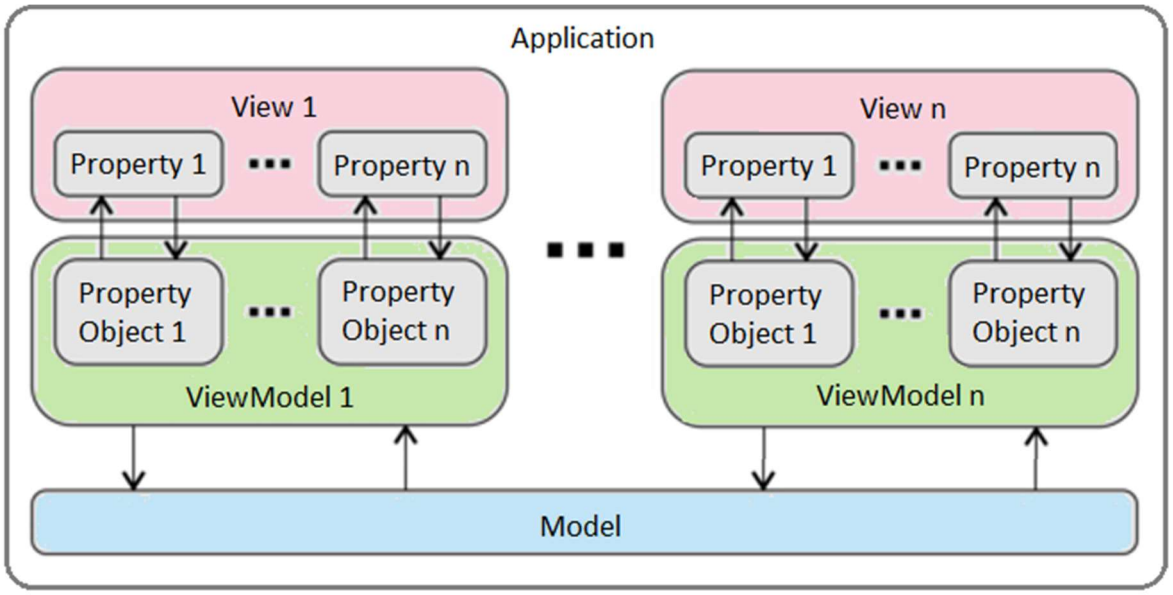
5.3 MVVM

Model-View-ViewModel on esiteltyjen muiden esitysmallien tapaan käyttöliittymien arkkitehtuurimalli, joka pyrkii eriyttämään esitys- ja applikaatiologiikat toisistaan.

MVVM-mallissa käyttöliittymä määritellään deklaratiivisesti ja siinä hyödynnetään sovelluskehyksen tiedonsidontaa näkymän yhdistämiseksi muihin kerroksiin. Sovelluskehyksen tiedonsidontainfra mahdollistaa käyttöliittymän löyhät kytkökset (loose coupling) pitäen sidotut tiedot synkronoituna ja reitittäen käyttäjän syötteet niille tarkoitettuihin komentoihin. (Microsoft 2017; Microsoft 2018b; Vice & Siddiqi 2012.)

Tiedonsidonnan tarjoamien löyhien kytkösten avulla syntyy vähemmän vahvoja riippuvuuksia erityyppisten ohjelmakoodien välille, tämän myötä yksittäisten ohjelmakoodien yksikköjen, kuten metodien, luokkien, kontrollien jne. muuttaminen helpottuu aiheuttamatta tarkoituksettomia sivuvaikutuksia muihin yksikköihin. Tämä kytköksien tai riippuvuuksien välttäminen on esimerkki aiemmin esitellystä ongelmien eriyttämisperiaatteesta. (Microsoft 2017; Microsoft 2018b; Vice & Siddiqi 2012.)

MVVM-mallia noudatettaessa sovelluksen käyttöliittymä ja sitä vastaava esittämissekä applikaatio-logiikka on jaettu kolmeen eri tekijään (kuvio 7): **Näkymään**, joka kapseloi käyttöliittymän ja siihen liittyvän logiikan, **näkymän tietomalliin**, joka kapseloi esityslogiikan ja esittämisen tilan, ja **tietomalliin**, joka kapseloi sovelluksen bisneslogiikan ja datan. (Microsoft 2017; Microsoft 2018b; Vice & Siddiqi 2012.)



Kuvio 7. Model-View-ViewModel-yleiskuva (Vice & Siddiqi 2012).

Mallia noudattamalla:

- Sovelluksen ulkoasu voidaan toteuttaa kirjoittamatta koodia, mikäli näkymä on toteutettu täysin merkintäkielellä (XAML).
- Kehittäjät voivat luoda yksikkötestejä tietomalleille ja näkymien tietomalleille erillään näkymästä. Näkymän tietomallin yksikkötestit voivat testata samoja toiminnallisuuksia, kuin mitä näkymän välitykselläkin olisi tarkoitus suorittaa.
- Jos tietomallista on olemassa implementaatio, joka kapseloi olemassa olevaa liiketoimintalogiikkaa, sen muokkaaminen saattaa olla riskialtista tai hankalaa. Tässä tilanteessa näkymän tietomalli toimii adapterina tietomalliluokille ja auttaa välttämään muutoksia tietomallien ohjelmakoodiin.
- Kehitysprosessin aikana ohjelmoijat ja graafiset suunnittelijat voivat työskennellä itsenäisesti ja yhtä aikaa omien osa-alueidensa parissa. Graafikko voi keskittyä näkymään samalla kun ohjelmoija voi keskittyä logiikkaa sisältäviin komponentteihin. (Microsoft 2017; Microsoft 2018b.)

5.3.1 Model

Kuten tähän mennessä esitellyissä muissakin arkkitehtuureissa, myös Model-View-ViewModel-mallissa tietomallin tarkoitus on kapseloida sovelluksessa käsiteltävä data.

Tietomalleja käytetään yleensä tietoyhteyden kapseloivien repositorioiden tai palvelujen (service) kanssa. Tietomallit ovat ulkoasuttomia luokkia, jotka kapseloivat sovelluksen datan. Sikäli tietomallin voidaan ajatella kuvastavan sovelluksen domainmallia, johon yleensä kuuluu applikaatio- ja validointilogiikkaa. Tiedonsiirto-oliot (DTO), yksinkertaiset oliot (POCO) ja entiteetit ovat esimerkkejä tietomalleista. (Microsoft 2017.)

5.3.2 View

Näkymä vastaa käyttäjän näkemän käyttöliittymän rakenteesta, asettelusta ja ulkoasusta. Mieluiten mahdollisimman suuri osa näkymästä on toteutettu merkintäkielellä (XAML, HTML), mutta aina se ei ole mahdollista. Esimerkiksi animaatiot on saatettava joutua toteuttamaan näkymän parina toimivaan logiikkatiedostoon (code behind) esimerkiksi JavaScriptillä tai C#-kielellä, koska merkintäkielet harvemmin kykenevät animaatioon. (Microsoft 2017.)

5.3.3 View model

Näkymän tietomallissa toteutetaan komentoja ja propertyja joihin näkymästä voidaan tehdä sidonta (ks. 5.3.4), jolloin näkymän tietomalli tiedottaa näkymää muutoksista muutosilmoitustapahtumilla. Propertyt ja komennot, joita näkymän tietomalli tarjoaa, määrittävät näkymän tarjoamat toiminnallisuudet, mutta näkymä vuorostaan määrittelee, kuinka ne esitetään. Näkymän tietomalli on myös vastuussa näkymän vuorovaikutusten koordinoinnista tarvittavien tietomallien kanssa. (Microsoft 2017; Blue Spire 2020f.)

Tyypillisesti näkymän tietomallilla on yhden suhde moneen tyyppinen relaatio tietomalleihin. Näkymälle saatetaan antaa suora pääsy ja yhteys tiedon sidonnalla näkymän tietomallin kautta tietomalliin. Tässä tapauksessa tietomallit täytyy suunnitella tiedonsidonta ja muutosten ilmoitustapahtumat mielessä pitäen. (Microsoft 2017.)

Näkymän tietomallien tehtävä on myös tarjota näkymälle tietomalleista saatava data helposti kulutettavassa muodossa, mikä yleensä toteutetaan sovelluskehysten sisäisten tiedonmuunnosmekanismien avulla, joita ovat esimerkiksi IValueConverter-rajapinnat .NET-kehitysalustassa ja Aurelian sidontamoottorin käyttämät ValueConverter-luokat. (Microsoft 2017; Blue Spire 2020f.)

Arvonmuuntimien sijoittaminen näkymän tietomalliin on hyvä idea, koska näkymän tietomalli tarjoaa propertyja joihin näkymästä voidaan sitoutua. Arvonmuunnin esitellään tarkemmin luvussa 6.

Näkymän tietomallissa voidaan esimerkiksi paremman esitettävyyden vuoksi luoda uusi property yhdistämällä kaksi muuta propertyä (kuvio 8).

```
class Henkilö
{
  string Etunimi { get; set; }
  string Sukunimi { get; set; }
  string Kokonimi => $"{Etunimi} {Sukunimi}";
}
```

Kuvio 8. Kokonimen muodostaminen merkkijonointerpoloinnilla C#-kielessä.

5.3.4 Tiedonsidonta (data binding)

Ilman tiedonsidontaa kehittäjä joutuu tyypillisesti kirjoittamaan ohjelmalogiikkaa datan näyttämiseksi, käyttäjän syötteiden vastaanottamiseksi ja syötteiden säilömiseksi tietolähteeseen. Edellä mainittujen tarpeiden toistuvuudesta ja työläydestä syntyi ajatus, että saattaisi olla suotavaa kapseloida nämä toimenpiteet käyttöliittymäsovelluskehykseen. Tästä ideasta syntyi tiedonsidonta. (Noyes 2006.)

Tiedonsidonta on mekanismi muistissa olevien olioiden liittämiseksi käyttöliittymällä esitettäviin kontrolleihin (control) ja tiedon näyttämisen ja muokkaamisen synkronoimiseksi. Sidonnalla voidaan tehdä liitos tietolähteen ja kontrollin välille ja jättää sidotun tiedon visualisointi kontrollerin vastuulle. Tämä johtaa esitysyksityiskohtien parempaan kapselointiin. (Noyes 2006; Microsoft 2018b; Microsoft 2018a.)

Yksisuuntaisessa sidonnassa tieto virtaa vain yhteen suuntaan. Sidotun tietolähteen propertyt ja niiden arvot asettuvat käyttöliittymä kontrollin propertyihin, mutta muutokset kontrollin sisäisiin arvoihin eivät automaattisesti heijastu takaisin tietolähteeseen. **Kaksisuuntaisessa** sidonnassa kontrolliin kohdistuvat muutokset heijastuvat myös tietolähteeseen. (Blue Spire 2020f; Microsoft 2018a; Noyes 2006.)

Tiedonsidonnan idea on siis abstrahoida tiedon asettamisen ja muutosten kuuntelun, sekä muutosten päivittämisen yksityiskohdat. Tiedonsidontatoiminnallisuuden toteuttaa ja tarjoaa useimmiten moderneista sovelluskehyksistä löytyvät sidontamoottorit. (Microsoft 2018a; Google 2021; Blue Spire 2020f; Knockoutjs 2021.)

6 AURELIA

Aurelia on käyttöliittymäsovelluskehys ja kokoelma moderneja JavaScript-moduuleja, jotka yhdessä toimivat tehokkaana alustana työpöytä- ja mobiilisovelluksien rakentamiseen. Sovelluskehyksen lähdekoodi on täysin avointa, saatavilla GitHubista ja se perustuu avoimiin verkkostandardeihin. (Blue Spire 2020b.)

Monoliittisen sovelluskehyksen sijaan Aurelia on jaettu kokoelmiin toimintokeskeisiä moduuleja. Moduuleihin kuuluvat mm. riippuvuusinjektio (dependency injection), tiedonsidonta (data binding), templating ja reititys (routing). (Blue Spire 2020b.)

Moduulit on toteutettu joko TypeScriptillä tai ECMAScriptillä. Vaikka moduulien käyttötarkoitusta ei ole rajattu, on niiden pääkäyttökohde selvästi sovelluksen käyttöliittymän toteutus. (Blue Spire 2020b.)

Sovelluskehystä kannattaa käyttää, koska sen tarjoamat ratkaisut ovat geneerisiä ja valmiiksi testattuja monissa käyttötapauksissa ja ympäristöissä. Vastuu ja päätösvalta sovelluskehyksellä toteutettavista toiminnallisuuksista siirtyy kehyksen tarjoajalle sen kuluttajan sijaan. (Sifuentes & Rojas 2018.)

Kolikolla on myös kääntöpuolensa ja ohjelmisto, joka riippuu toisesta ulkopuolisesta ohjelmistosta toimiakseen, saattaa pahimmassa tapauksessa periä käytettävässä ohjelmistossa tapahtuvat haittavaikutukset. Esimerkiksi, jos käytettävästä ohjelmistosta paljastuu tietoturva-aukko, tai käytettävässä ohjelmistossa tapahtuu rikkovia muutoksia, saattavat nämä vaikuttaa myös sitä hyödyntävään ohjelmistoon. (Sifuentes & Rojas 2018.)

Aurelia kannattaa valita, koska se on varsin kattava verkkokäyttöliittymäsovelluskehys. Edellä mainittujen moduulien lisäksi Aureliaan on saatavilla liitännäisiä mm. kansainvälistämiseen, validointiin ja käyttöliittymän virtualisointiin. Sen mukana tulee myös vahva komentoliittymä, joka auttaa suuresti projektien luomista ja kehittämistä. (Blue Spire 2020b.)

Aurelia on rakennettu joustavaksi ja sen tarjoamat ominaisuudet ovat myös korvattavissa. Esimerkiksi, jos on tarve käyttää sovelluskehyksen tarjoaman sijaan jotain

toista templating engineä, voi vakiototeutuksen helposti konfiguroida pois käytöstä. (Blue Spire 2020b.)

Aurelia lupaa nopeat renderöintiajat ja tehokkaan muistinhallinnan. Sovelluskehys noudattaa semanttista versiointia (semver) ja on ollut jatkuvan kehityksen alla ilman rikkovia muutoksia ydinrajapintaan sitten version 1.0, joka julkaistiin 27.7.2016. Joten Aureliasta voidaan todeta, että se on kehittäjille, jotka arvostavat ohjelmointirajapintojen vakautta.(Blue Spire 2020b.)

Aureliassa komponentit on mahdollista toteuttaa puhtaalla (vanilla) JavaScriptillä tai TypeScriptillä. Se on kehittäjille, jotka suosivat helposti opittavia ja muistettavia ohjelmointimalleja. Sen yksinkertaiset ja yhdenmukaiset konventiot auttavat kehittäjää tuottamaan vankkaa koodia ja vähentämään sen ylläpitämiseen kuluvaa aikaa. Konventioiden avulla Aurelia tavoittelee, että kehittäjien aika voitaisiin kohdistaa itse lopputuotteeseen kehitysalustan sijaan. (Blue Spire 2020b.)

Aurelia noudattaa Web-standardeja, joten sillä toteutettuihin projekteihin on helposti integroitavissa myös muita kolmansien osapuolien sovelluskehyksiä ja kirjastoja, kuten jQuery, Boostrap, React jne. (Blue Spire 2020b).

6.1 Komponentit

Aureliassa käyttöliittymäkomponentit koostetaan parista näkymiä ja näkymän tietomalleja. Näkymät toteutetaan HTML-merkintäkielellä ja ne renderöityvät selaimen dokumenttirakenteeseen (DOM).

Näkymän tietomallit toteutetaan ES Nextillä, joka tarkoittaa toteutushetkellä uusinta ECMAScript versiota. Näkymän tietomallin tehtävänä on tarjota tietoa ja toiminnallisuuksia näkymälle.

Aurelian templating engine käyttää riippuvuusinjektiota yhdistämään komponentin osat suorituksen aikana. Käyttöliittymäkomponentin luomiseen tarvitaan kaksi tiedostoa: yksi näkymälle ja toinen sen logiikkatiedostolle. Seuraavassa kuviossa on esimerkki yksinkertaisesta Hello-komponentista.

```
<template>
  <input value.bind="firstName">
  <input value.bind="lastName">

  <button click.trigger="sayHello()">Say Hello</button>
</template>
```

Kuvio 9. Hello-komponentin näkymä.

Kuviossa 9 on luotu näkymä komponentille. Kuviossa huomattavaa on, että näkymä kääritään web-komponenttien HTMLTemplateElement-elementtiin. Aurelia käyttää näkymissään siis standardin mukaisia HTML-templateja.

Kuviosta nähdään myös Aurelian helposti omaksuttava sidontasyntaksi. Yksinkertaisimmillaan sidonta tapahtuu lisäämällä minkä tahansa dokumenttirakenteen HTML-attribuutin perään `.bind="someProperty"` ja sovelluskehys hoitaa loput. Se siis etsii näkymän tietomallista parametria vastaavaa propertya ja sen löydyttyä sitoo attribuutin siihen.

```
export class Hello {
  constructor() {
   this.firstName = 'Joni';
   this.lastName = 'Neutroni';
  }

  sayHello() {
   alert(`Hello ${this.firstName} ${this.lastName}!`);
  }
 }
```

Kuvio 10. Hello-komponentin näkymän malli.

Kuviossa 10 nähdään komponentin näkymän malli. Merkittävää näkymän mallissa on se, että pohjimmillaan se on tavallinen (vanilla) JavaScript-luokka. Tässä tämä tarkoittaa, että näkymän malli ei riipu lainkaan sovelluskehyksestä. Täten esimerkiksi sen kehittäminen ei edellytä lainkaan sovelluskehyksen tuntemusta. Myös logiikkatiedoston lähdekoodia voidaan tuoda tai käyttää muuallakin, kuin Aurelia-sovelluskehyksen näkymän tietomallissa.

6.2 Elinkaarimetodit

Muodostaja on metodi, jota käytetään alustamaan luokasta instanssi. Aurelian näkymän tietomallit perustuvat luokkiin, joten niillä on muodostaja. Mikäli konstruktoria ei erikseen määritetä, olio-ohjelmointikielten konventioiden mukaan oletusmuodostajaa käytetään. (Tutorials Point 2021.)

Kiinnittymis-metodia (attached) kutsutaan, kun komponentti kiinnittyy dokumenttirakenteeseen (DOM) eli se renderöidään. **Irtaantumis**-metodi (detached) vuorostaan ajetaan, kun komponentti lakkaa näkymästä, eli poistuu dokumenttirakenteesta. (Tutorials Point 2021.)

Komponenttien tapahtumiin liittyvä logiikka tulee lähdekoodissa sijoittaa ja ohjelman suorituksessa ajoittaa elinkaarimetodeihin. Esimerkiksi, jos komponentin elinkaareen liittyy resursseja, jotka on vapautettava esimerkiksi näkymän suljettaessa, voidaan tämä logiikka sijoittaa irtaantumismetodiin. (Tutorials Point 2021.)

6.3 Templatemoottori

Aiemmin esiteltiin että, HTML-merkintäkielessä on `<template>`-elementti, jota Aureliassa käytetään merkitsemään templating enginen renderöitävä näkymä. Yksittäinen template voi edustaa vaikka kokonaista sivua, tai se voi olla pienempi kokonaisuus, kuten omatekoisen käyttöliittymäkontrollin näkymä.

Aurelia noudattaa template-elemetin käyttöön liittyviä standardeja ja tekee niiden käytöstä erittäin helppoa. Moottori on helppo oppia, mutta silti tarpeeksi kattava toteuttamaan jopa monimutkaisimmankin sovelluksen (Blue Spire 2020c).

```
<template>
  <h1>Hello world.</h1>
</template>
```

Kuvio 11. Yksinkertainen template.

Kuviossa 11 on luotu yksinkertainen template, jota voidaan luonnehtia staattiseksi, sillä siihen ei liity toiminnallisuutta, joka voisi vaikuttaa sen sisältöön, eli se on muuttumaton. Vaihtoehtoisesti sitä voitaisiin kutsua myös tilattomaksi tai tyhmäksi komponentiksi.

6.4 Merkkijonointerpolointi

ECMAScript 6 -spesifikaatio toi JavaScriptiin template-literaalit ja merkkijonointerpoloinnin. Template-literaalit ovat merkkijonoliteraaleja, jotka mahdollistavat lausekkeiden (expression) upottamisen merkkijonoon. (Ecma International 2015; Mozilla 2021b.)

Aureliassa vastaavien ominaisuuksien syntaksi on täsmälleen sama. Kuviossa 12 nähdään kuvion 11 templaatti muutettuna tervehtimään käyttäjää nimellä.

```
<template>
  <h1>Greetings ${name}!</h1>
</template>
```

Kuvio 12. Template-literaali ja merkkijonon interpolointi.

Kuviossa 12 nähdään merkkijonon interpolointi-vittaus propertyyn `name`. Templatea renderöitäessä moottori etsii näkymän tietomallista viitettä vastaavaa propertyä ja asettaa interpolointisyntaksin tilalle viitteen arvon merkintäkielellä.

Merkittävintä Aurelian template-literaaleissa on kuitenkin se, että näkymässä käytettävien upotettujen lausekkeiden arvojen muuttuessa näkymä renderöityy automaattisesti uusilla arvoilla.

6.5 Toistin

Template-moottorin yksi merkittävimmistä ominaisuuksista on kyky ilmaista dokumenttirakenteessa toistuvia elementtejä tehokkaasti. Seuraavaksi esitetään näkymä, jossa käytetään toistorakennetta (kuvio 13).

```
<template>
  <p repeat.for="i of 10">${10-i}</p>
  <p>Blast off!</p>
</template>
```

Kuvio 13. Toistimen käyttö HTML-elementtiin.

Kuviossa 13 ohjelmaa suoritettaessa edellä nähty lauseke muuntuu dokumenttirakenteessa kuvion 14 muotoon.

```
<view-name>
  <p>10</p>
  <p>9</p>
  <p>8</p>
  <p>7</p>
  <p>6</p>
  <p>5</p>
  <p>4</p>
  <p>3</p>
  <p>2</p>
  <p>1</p>
  <p>Blast off!</p>
</view-name>
```

Kuvio 14. Lähtölaskenta-toistimen luoma dokumenttirakenne suorituksen aikana.

Tuloksena saadaan siis lähtölaskentaa esittävä joukko `<p>`-elementtejä käärittynä `<view-name>`-elementtiin. Kääre on todellisuudessa Aurelian custom element, joita käsitellään HTML-behavioreiden yhteydessä.

Muiden verkkokäyttöliittymäsovelluskehysten tapaan Aurelian toistolauseke toimii pääasiassa iteroitavilla kokoelmilla, mutta poikkeuksellisesti Aurelia tukee myös kokonaislukuja, joita juuri hyödynnettiin lähtölaskentaesimerkissä.

6.6 Tiedonsidonta

Tiedon synkronointi näkymän ja näkymän mallin välillä on aina ollut eräs kipukohta verkkosovellusten käyttöliittymien toteuttamisessa perus verkkotekniikoilla kuten HTML- ja JavaScript-kielillä.

MVVM-mallin yhteydessä on esitelty tiedonsidonta, mutta tässä esitetään Aurelian kontekstissa sovelluskehyksen tarjoamat sidontakomennot.

`one-time` on sidontakomento, jossa tieto asettuu näkymän tietomallista näkymään vain kertaalleen. Kertasidonta on hyödyllinen, kun tiedossa on, että sidottu tieto ei tule muuttumaan sovelluksen käynnissä olon aikana.

```
<a href.one-time="homeUrl"></a>
```

`to-view` on näkymään päin tietoa kuljettava sidontakomento, jossa muutokset kulkevat vain yhteen suuntaan, näkymän tietomallista näkymään.

```
<a href.to-view="newsUrl">Latest News</a>
```

Kaksisuuntaisessa `two-way`-sidonnassa muutokset komponentin ns. etu- että takapäässä heijastuvat aina vastakappaleeseen. Useimmiten tätä sidontakomentoa käytetään, kun halutaan kaapata käyttäjän syötteitä käyttöliittymän näkymästä.

```
<input type="email" value.two-way="email">
```

Komennoista yleisin on kuitenkin `bind`, jossa sidontatyyppi määräytyy automaattisesti sidottavan attribuutin mukaan. Lomakkeissa käytettäville syötekentille tämä tarkoittaa sidontaa molempiin suuntiin.

```
<input type="password" value.bind="password">
```

6.7 Sidonnan käytös (binding behavior)

Binding behavior on yksi kategoria näkymäresursseja (view resource), joihin lukeutuvat mm. custom attribuutit ja -elementit sekä arvomuuntimet. Binding behaviorin kanssa samankaltaisin näkymäresurssi (view resource) on arvomuunnin, näitä molempia käytetään deklaratiivisissa sidontalausekkeissa vaikuttamaan sidonnan toimintaan. (Blue Spire 2020e.)

Merkittävin ero näiden kahden välillä on kuitenkin se, että binding behaviorilla on täysi pääsy sidontainstanssiin koko sen elinkaaren ajan. Arvomuunnin kykenee ainoastaan vaikuttamaan arvoon sen ylittäessä rajan mallista näkymään tai toisinpäin. (Blue Spire 2020e.)

Aureliasta löytyy muutamia valmiita binding behavioreja yleisimpiin käyttötapauksiin, joista tarkastellaan seuraavaksi kuristinta (throttle). Kuristimella voidaan rajoittaa, kuinka usein tiedon päivityksiä halutaan suorittaa, eli esimerkiksi kuinka usein käyttäjän syötteestä näkymän kautta viedään muutokset näkymän tietomalliin. (Blue Spire 2020e.)

```
<input type="text" value.bind="someProperty & throttle">
```

Kuvio 15. Throttle binding behavior

Symbolia `&` käytetään sidontalausekkeessa merkitsemään binding behavioria. Behaviorille voidaan myös antaa parametreja syntaksilla, jossa ensin määritetään behaviorin tunniste ja sitten parametrin arvo erotettuna kaksoispisteellä (`:`). Parametrejä voidaan antaa niin monta ja siinä järjestyksessä, kun käytettävän binding behaviorin rajapinta odottaa. (Blue Spire 2020e; Duffield 2018.)

Debounce binding behavior on miltei samanlainen kuin kuristin, mutta kuristimesta eroten debounce odottaa, kunnes edellisestä muutoksesta on kulunut tietty aika. Ellei debouncelle erikseen parametrina määritetä päivitysviivettä, se vakioituu kahteen sataan millisekuntiin (200 ms). (Blue Spire 2020e; Duffield 2018.)

Debounce sopii erityisen hyvin esimerkiksi tilanteeseen, jossa olisi suotavaa odottaa, että käyttäjä lopettaa syötekenttään kirjoittamisen sen sijaan, että jokainen näppäinpainallus aiheuttaisi muutostaptahtuman (kuvio 16).

```
<input type="text" value.bind="someOtherProperty & debounce:1000">
```

Kuvio 16. Debouncen käyttö parametrilla.

6.8 Arvomuunnin (value converter)

Arvomuuntimet ovat syntaktisesti hyvin samanlaisia binding behavioreiden kanssa. Sidontalausekkeen yhteydessä käytetään pipe-operaattoria | merkitsemään arvomuunninta. Arvomuuntimien tehtävä on toimia komponentin osien välissä ja muotoilla sidottua tietoa.

```
<template>
  <require from="./time-ago"></require>
  call duration ${callStartedTimestamp | timeAgo}
</template>
```

Kuvio 17. TimeAgo-muuntimen käyttö näkymässä.

Kuviossa 17 nähdään näkymässä sidontalausekkeessa muuttuja modifiedDateTime, joka piiputetaan arvomuuntimeen timeAgo.

```
import moment from 'moment';

export class TimeAgoValueConverter {
  toView = date => moment(date).fromNow();
}
```

Kuvio 18. TimeAgo-muuntimen logiikka.

Käytännössä TimeAgoValueConverter-luokan toView-propertya kutsutaan piiputetulla aikaleimalla, joka vuorostaan kutsuu moment-kirjastoa. Moment-kirjaston `fromNow`-metodi suorittaa vertailun kutsun kohteena olevan Moment-tyypin aikaleiman ja käyttäjän laitteen sisäisen kellon nykyhetken välillä ja palauttaa tuloksen stilisoituna laitteen lokalisointiasetusten mukaan (kuvio 18).

6.9 HTML Behavior

Aurelian termi *HTML Behavior* on yläkäsite: *Custom elementeille* ja *Custom attribuuteille*. Behaviorit ovat perusteellinen osa miltei jokaista Aurelia-sovellusta ja ne auttavat luomaan komponentisoituja sovelluksia mahdollisimman uudelleenkäytettävällä ohjelmakoodilla. (Blue Spire 2020a.)

Behavioreilla on mahdollisuus laajentaa HTML-kielen toiminnallisuuksia. Custom-elementit ovat omatekoisia HTML-elementtejä, joilla on yleensä näkymä, joka kuvannetaan sovelluksessa osana sivua. Custom-attribuuteilla voidaan vaikuttaa vakio HTML-elementtien, kuten divien tai buttonien toiminnallisuuteen. (Blue Spire 2020a.)

Suorituksen aikana Aurelia päättelee sovelluskehyksen konventioiden ja nimeämiskäytäntöjen mukaan mitkä näkymälogiikka- ja toimintalogiikkatiedostot kuuluvat liittää yhteen. Nimeämiskäytäntö JavaScript-luokille on PascalCase, jossa jokainen sana alkaa isolla alkukirjaimella (Mozilla 2021a). HTML-kieli ei erottele merkkejä kirjainkoon mukaan ja se useimmiten normalisoidaan pieneen kirjasinkokoon (WHATWG 2021).

Käytännössä Aurelia karsii luokan nimen lopusta luokan toteuttavan toiminnallisuuden nimen ja loppuliitteen kuten CustomElement ja muuttaa luokan nimen kirjoitusasun *dash-caseksi* liittääkseen näkymä- ja logiikkatiedostot toisiinsa niiden nimien kirjasinasujen erosta huolimatta (Blue Spire 2020d). Dash-case tunnetaan myös nimellä *kebab-case*.

Esimerkiksi siis *HelloWorldCustomElement*-niminen luokka muuntuu muotoon *hello-world* viitattaessa siihen näkymän puolella. Kuviossa 19 nähdään kuinka näkymästä viitataan hello-world-komponenttiin.

```
<template>
  <require from="hello-world"></require>
  <hello-world></hello-world>
</template>
```

Kuvio 19. Custom-elementin käyttö templatessa.

Ennen kuin omatekoista elementtiä voi upottaa toiseen näkymään on se tuotava templaten näkyvyysalueelle (scope) `require`-elementillä. Elementin attribuutti `from`, vastaanottaa parametrina polun upotettavaan elementtiin.

Polku voidaan määrittää kahdessa eri muodossa, joko relatiivisena sovelluksen juuresta alkaen tai alkaen nykyisen näkymän sijainnista. Kuviossa 19 polkuviittaus alkaa sovelluksen juuresta. Polku määritetään alkavaksi nykyisen näkymän sijainnista lisäämällä polun alkuun joko `./` tai `../`.

7 BOOTSTRAP

Bootstrap on yksi suosituimmista avoimen lähdekoodin sovelluskehyksistä verkkokäyttöliittymien ulkoasun luomiseen. (Marah & Jakobus 2018). Vaihtoehtoisia käyttöliittymäkirjastoja Bootstrapille ovat mm. Semantic UI ja Bulma.

Bootstrapin suosion nopeaan kasvuun vaikuttavia tekijöitä ovat luultavasti mm. sen veloitukseton ja rajoittamaton saatavuus ja käyttö niin yksityiseen, opetukselliseen tai liiketoiminnalliseen käyttöön (Marah & Jakobus 2018).

Ainakin yhden käyttöliittymän ulkoasuun keskittyvän sovelluskehyksen tunteminen on ehdottomasti tärkeä taito ja toimii tehokkaana lisänä kenen tahansa web-kehittäjän työkalupakissa (Marah & Jakobus 2018; Lambert & Marah 2017).

7.1 Mobiilikokemuksen priorisoiminen

Bootstrap on lähtökohtaisesti suunniteltu toimimaan päätelaitteen resoluutiosta tai ruudun koosta riippumatta, niin puhelimilla kuin pöytäkoneillakin. Mobiili ensin -suunnitteluperiaate velvoittaa, että komponenttien on ulkoasun ja toimintojen on säilyttävä ennallaan myös pienimmillä ruudunkoilla. (Marah & Jakobus 2018; Lambert & Marah 2017.)

Syy suunnitteluperiaatteen takana on varsin yksinkertainen. Kuvitellaan tilanne, jossa ollaan kehittämässä verkkosivua huomioimatta vaihtoehtoisia ruudun kokoja. Tällaisessa tapauksessa ruututila mitä todennäköisimmin tulee ahdetuksi täyteen listoja, taulukkoja, kontrolleja jne. (Marah & Jakobus 2018.)

Pahimmassa tapauksessa asettelun joustamattomuudesta koituvat käytettävyysongelmat ilmenevät vasta palautteena loppukäyttäjiltä. Mikäli todetaan, että lopputuotteen on tuettava myös rajoittavampia ominaisuuksia, kuten puhelimen ruutukokoa tai kuvasuhdetta ja käyttöliittymän virheellisesti oletettu ruutuala on jo ahdettu täyteen, tarkoittaa se mitä todennäköisimmin, että on palattava takaisin suunnittelupöydän ääreen, sillä paljon asioita liian pienessä tilassa on huomattavasti hankalampi esittää, kuin vähän asioita liian suuressa tilassa. Siksi monet käyttöliittymäkirjastot, kuten tässä tapauksessa Bootstrap kannattavat alhaalta ylöspäin -lähestymistapaa,

jossa kehittäjät joutuvat huomioimaan rajoittavimman tekijän ennen käyttöliittymän laajentamista. (Marah & Jakobus 2018.)

7.2 Ruudukkojärjestelmä

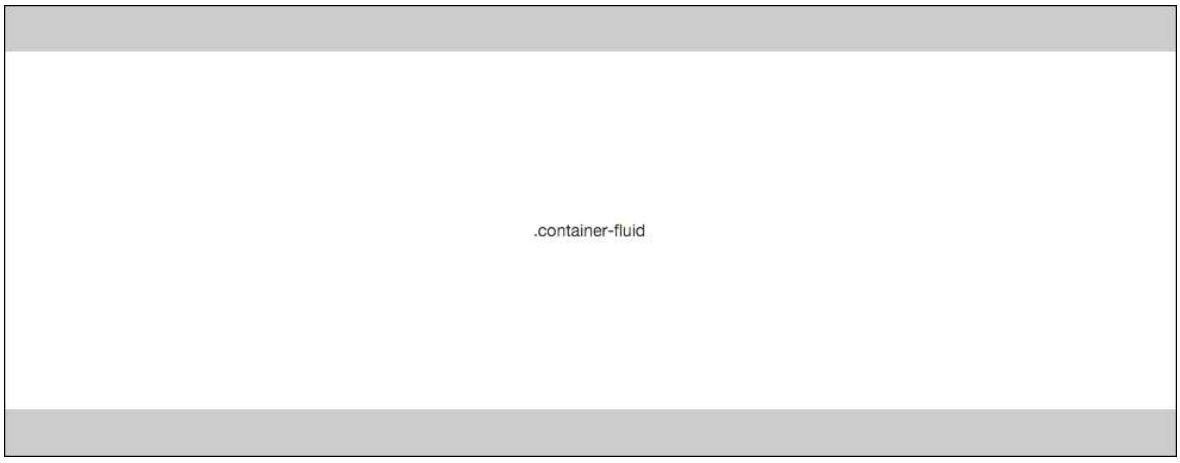
Käyttöliittymän esitysasun päätelaiteriippumattomuutta kutsutaan myös responsiivisuudeksi. Bootstrapin responsiivisuus perustuu sen ruudukkojärjestelmään, johon mm. kirjaston tarjoamia komponentteja voidaan asettaa. (Lambert & Marah 2017.)

Bootstrapin ruudukkojärjestelmä on rakenne, joka koostuu kolmesta erillisestä, mutta perusteellisesti toisiinsa sidonnaisista osista: kaiken kapseloivasta **containerista**, joka jakautuu vaakasuuntaisiksi **riveiksi**, jotka vuorostaan jakautuvat tasaisesti kahdeksitoista (12) pystysuuntaiseksi **sarakkeeksi** (Marah & Jakobus 2018).

7.3 Kontti

Kontti ovat sovelluksen perusteellisin asetteluelementti ja se on edellytys sovelluskehyksen ruudukkojärjestelmän käyttämiseen. Kontteja käytetään esittämään sisältöä, luomaan tyhjää tilaa sisällön ympärille (padding) ja joskus keskittämään sisältöä. Kontteja on mahdollista asettaa sisäkkäin, mutta useimmiten se on tarpeetonta. (Lambert ym. 2017.)

Kontin voi määrittää kahdella tapaa, jotka esitellään seuraavissa kuvioissa.



Kuvio 20. Koko ruudun leveydellään täyttävä kontti.

Kuviossa 20 nähdään **.container-fluid**-kontti, joka venyy täyttämään koko selainikkunan koon vaakasuunnassa.

Kuvio 21. Kontti, jonka leveys määräytyy kirjaston pysäytyspisteiden mukaan suhteessa ruudun kokoon.

Kuviossa 21 nähdään **.container**, jonka leveys muuttuu responsiivisten pysäytyspisteiden (breakpoint) välillä ruudun koon muuttuessa. Pysäytyspisteillä toimivan kontin käyttö on suositeltavaa esimerkiksi luettavan sisällön esittämiseen (Lambert ym. 2017).

Verkkokäyttöliittymissä pysäytyspiste tarkoittaa määrättyjä ruudun dimensoita joiden mukaan tyylittelysäännöt muuttuvat. Ruudun koon muutosta seurataan mediakyselyillä, jotka määrittävät tyypillisesti viewportin leveyden mukaan mitkä mediasäännöt ovat voimassa. Esimerkiksi säännön `@media (min-width: 768px)` mukaiset tyylit astuvat voimaan ruudun leveyden ollessa vähintään 768 pikseliä levelä. (Lambert & Marah 2017; Marah & Jakobus 2018.)

Rivit (row) ovat kääreitä (wrapper) sarakkeille. **Sarakkeilla** (col) on vaakasuuntainen sisennys eli palstaväli (gutter) sarakkeiden välissä olevan tilan hallintaan. Riviluokalle on vakiona määritelty negatiivinen marginaali, joka kumoaa sarakkeessa määritellyn sisennyksen. Näin sarakkeiden sisältö tasoittuu pystysuunnassa vasempaan reunaan. Ruudukkoasettelussa sisältö täytyy asettaa sarakkeiden sisään ja vain sarakkeet voivat olla rivien välittömiä jälkeläisiä (immediate children). (Twitter 2018b.)

Sarake luodaan antamalla `div`–elementille luokka syntaksilla: `col-{breakpoint}-{fraction}`, josta breakpoint on valinnainen pysäytyspiste ja fraction vuorostaan sarakkeen leveys murto-osana koko ruudun leveyttä.

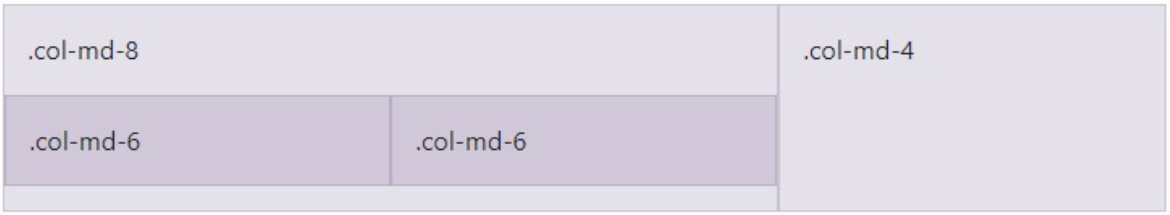
Esimerkiksi kun halutaan kolme keskenään saman levyistä saraketta, jotka täyttävät kaiken saatavilla olevan tilan vaakasuunnassa, asetetaan elementin luokkamäärittelyyn leveydeksi 4 (kuviot 21 ja 22).

Kuvio 21. Kolme ruudun koosta riippumatta saman levyistä saraketta.

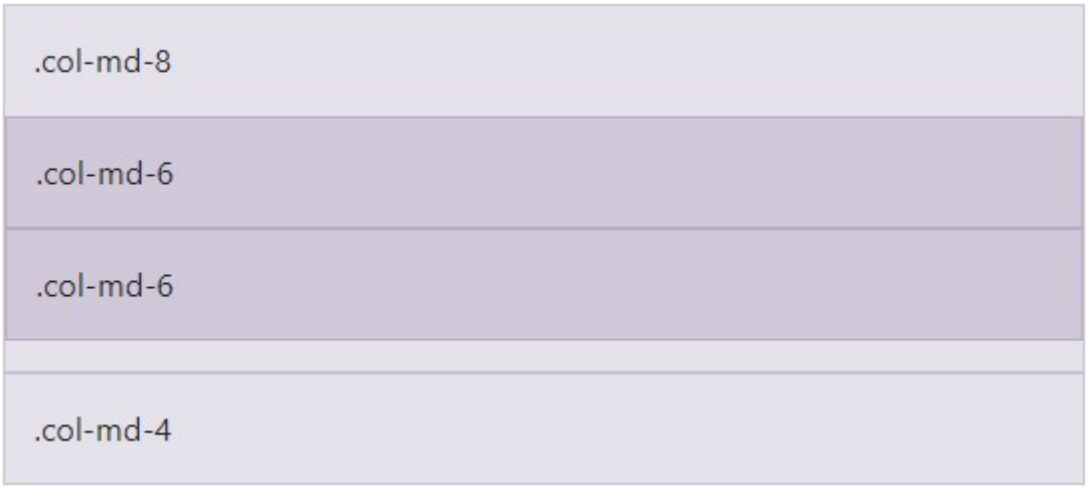
```
<div class="row">
  <div class="col-4">.col-4</div>
  <div class="col-4">.col-4</div>
  <div class="col-4">.col-4</div>
</div>
```

Kuvio 22. Kuvion 21 asetelman toteutus.

Kuvioissa 23–25 on esitetty ruudukkoasettelu, jossa ruudun leveyden ollessa medium-pysäytyspistettä, suurempi näytetään sisältöä kahdessa rinnakkaisessa sarakkeessa. Vasemman sarakkeen sisällä on rivi, jolla on myös kaksi rinnakkaista medium-pysäytyspisteen saraketta. Kuviossa 24 nähdään ruudun leveyden pienentyessä medium-pysäytyspistettä pienemmäksi, että sarakkeet venyvät täyttämään koko ruudun leveystilan ja asettuvat pystysuunnassa samaan järjestykseen, kuin miten ne kuvion 25 rakenteessakin on määritetty.



Kuvio 23. Ruudukkoasetelma tarkasteltuna desktop-koossa, jossa vasempaan sarakkeeseen upotettuna kaksi samankokoista saraketta samalla rivillä.



Kuvio 24. Kuvion 23 ruudukkoasetelma tarkasteltuna mobiilinäkymässä.

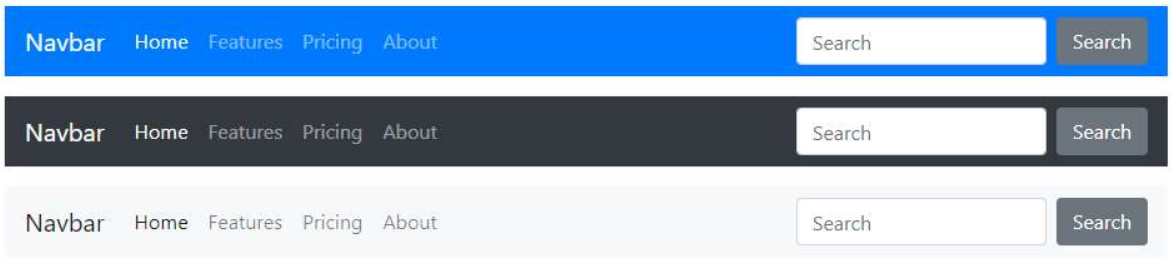
```
<div class="row mb-3">
 <div class="col-md-8">
  <div class="pb-3">
   .col-md-8
  </div>
  <div class="row">
   <div class="col-md-6">.col-md-6</div>
   <div class="col-md-6">.col-md-6</div>
  </div>
 </div>
 <div class="col-md-4">.col-md-4</div>
</div>
```

Kuvio 25. Kuvioiden 23 ja 24 responsiivisen asettelun toteutus.

7.4 Yleisimpiä komponentteja

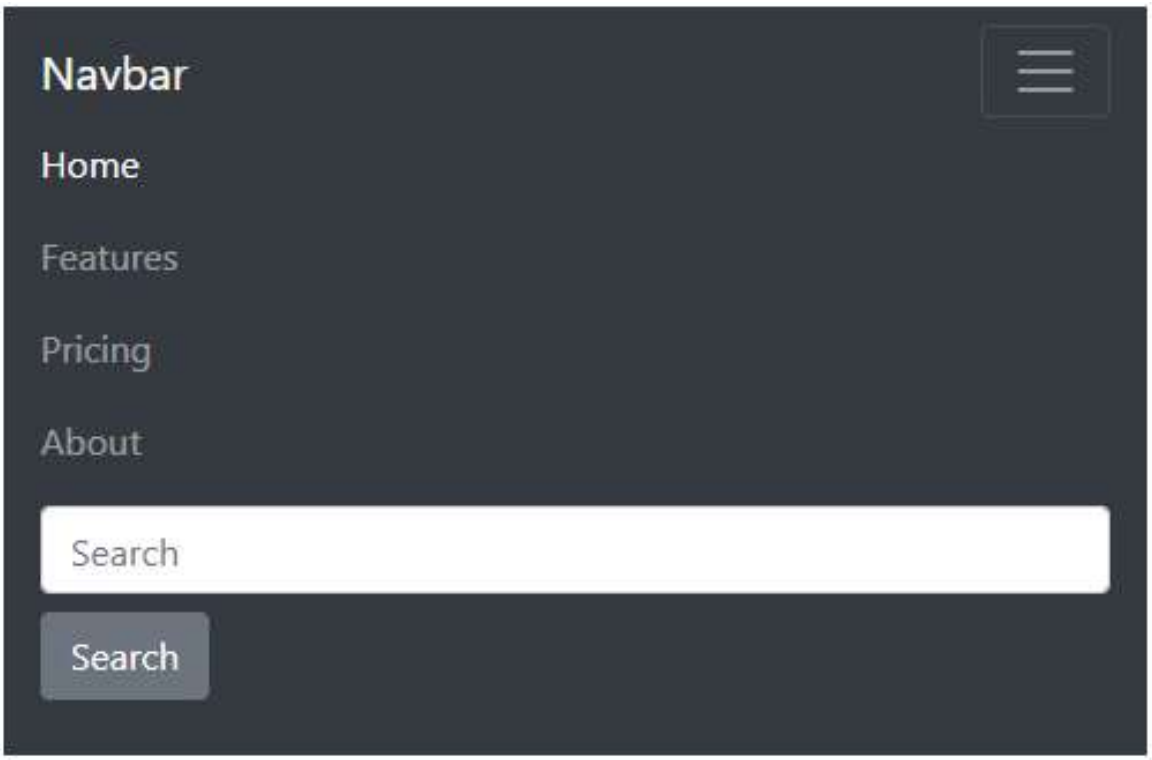
Seuraavaksi käsitellään muutama Bootstrapin yleisimmistä valmiista komponenteista, joita yhdistelemällä saadaan katettua suurin osa yksinkertaisen verkkokäyttöliittymän perustoiminnallisuuksista.

7.4.1 Navigointipalkki



Kuvio 26. Navigointipalkki ja sen vakiovärit .

Navigointipalkki (navbar) on yksi Bootstrapin tunnetuimmista ominaisuuksista sen toiminnan takia. Selainikkunan kutistuessa navigointipalkki tiivistyy ruututilan vähentyessä. (Hong 2018.)



Kuvio 27. Mobiilinäkymää varten kutistunut tumma navigointipalkki, jonka linkit vedetty esiin hampurilaisvalikosta.

Kuviossa 27 nähdään, että ruudun kutistuttua riittävästi, piiloutuvat navigointipalkin elementit hampurilaisvalikon taakse. Kohteet saadaan liukumaan näkyville valikkonäppäintä napauttamalla. Valikkonäppäin löytyy kuvan oikeasta yläkulmasta. (Hong 2018.)

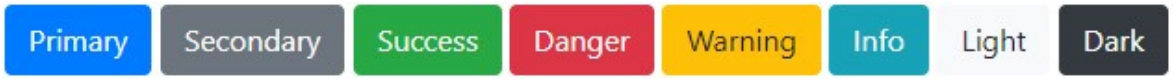
```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
 <a class="navbar-brand"
  href="#">Navbar</a>

 <button class="navbar-toggler"
    type="button"
    data-toggle="collapse"
    data-target="#navbar">
  <span class="navbar-toggler-icon"></span>
 </button>

 <div class="collapse navbar-collapse"
   id="navbar">

  <ul class="navbar-nav mr-auto">
   <li class="nav-item active">
    <a class="nav-link"
     href="#">Home</a>
   </li>
   <li class="nav-item">
    <a class="nav-link"
     href="#">Features</a>
   </li>
   <li class="nav-item">
    <a class="nav-link"
     href="#">Pricing</a>
   </li>
   <li class="nav-item">
    <a class="nav-link"
     href="#">About</a>
   </li>
  </ul>

  <form class="form-inline my-2 my-lg-0">
   <input class="form-control mr-sm-2"
      type="text"
      placeholder="Search">
   <button class="btn btn-secondary my-2 my-sm-0"
      type="submit">Search</button>
  </form>

 </div>
</nav>
```

Kuvio 28. Responsiivisen navigointipalkin merkintäkoodi kokonaisuudessaan.

Kuviosta 28 nähdään, että navigointipalkkiin on asetettu nappi luokalla `navbar-toggler`, joka on näkyvissä vain kun navigointipalkki kutistuu mobiilinäkymää varten. Napille on myös määritelty attribuuteilla `data-toggle` ja `data-target` toimenpide ja

kohde, johon napin painallus vaikuttaa. Nappia painettaessa kirjaston koodi saa aikaan, että kohde-attribuutissa määritetty elementti `#navbar` avautuu tai kutistuu.

7.4.2 Tyyliluokat ja painike



Kuvio 29. Tyyliluokin koristellut bootstrap-painikkeet.

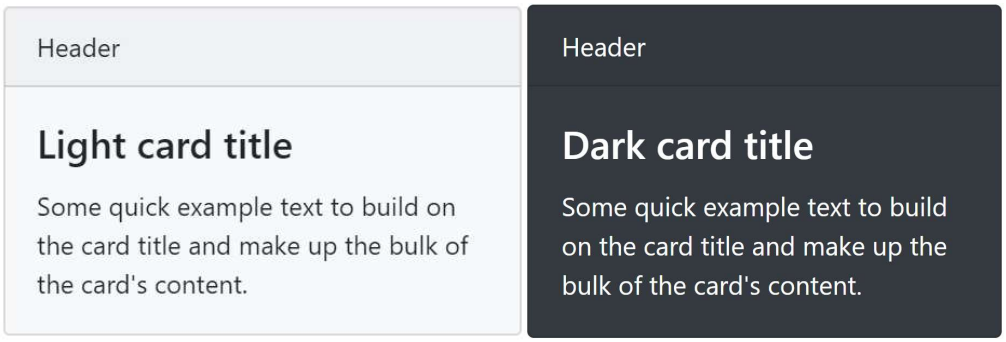
Kirjastosta löytyy komponenteille lukuisia semanttisia tyylimäärittelyjä, joilla voidaan viestiä komponentin merkitystä. Tyyliluokkien nimet ovat melko itseselitteisiä, mutta esimerkiksi tyyliasulla `primary` voidaan käyttöliittymässä viestiä ensisijaista toiminnallisuutta, kuten esimerkiksi lomakkeessa Tallenna-painike. Lomakkeessa toissijainen toiminnallisuus (`secondary`) voisi olla esimerkiksi Peruuta-painike. (Twitter 2018a.)

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
```

Kuvio 30. Kuviossa 29 nähtävien painikkeiden rakenne.

Kirjaston tyyliluokat toimivat myös suurimmaksi osaksi muidenkin kirjaston tarjoamien komponenttien kuin napin kanssa.

7.4.3 Kortti



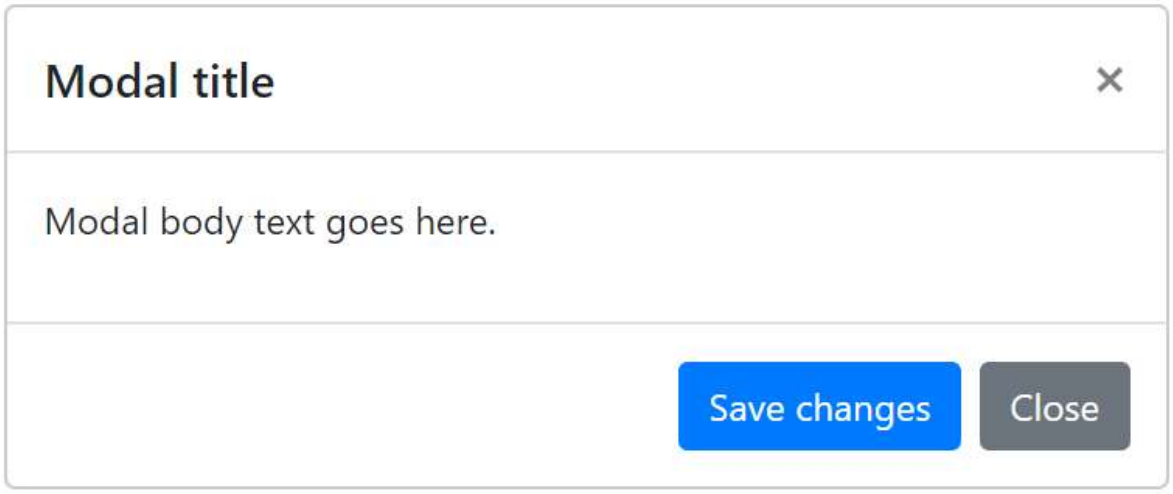
Kuvio 31. Tumma- ja vaaleateemaiset kortit.

Kortti (card) on monipuolinen ja joustava sisällön koostaja. Korttiin kuuluu valinnaiset ylä- (header) ja alatunnisteet (footer). Kortissa voidaan esittää esimerkiksi kuvia, tekstiä, listaryhmiä, linkkejä jne. Kortti pohjautuu `card-body`-luokan ympärille, jota käytetään aina kun halutaan näyttää sisennettyä sisältöä kortissa. Vakiona korttien leveys täyttää kaiken mahdollisen horisontaalisen tilan, ellei leveyteen erikseen vaikuteta. (Twitter 2018c.)

```
<div class="card text-white bg-dark">
 <div class="card-header">Header</div>
 <div class="card-body">
  <h4 class="card-title">Dark card title</h4>
  <p class="card-text">Some quick example text
   to build on thecard title and make up
   the bulk of the card's content.</p>
 </div>
</div>
```

Kuvio 32. Lähdekoodi tumman kortin luomiseen.

7.4.4 Modaali



Kuvio 33. Bootstrapin vakiomodaali.

Modaalit ovat tyypillisesti ponnahdusikkunoita, joka ilmestyy avoinna olevan näkymän eteen uuden välilehden tai ikkunan avautumisen sijaan. Ne saattavat himmentää taustalle jättämänsä näkymän vetääkseen enemmän huomiota modaaliin. Modaaleilla voidaan käyttäjälle näyttää ilmoituksia aiheuttamatta sivuston uudelleenlatauksia tai siirtymiä ja näin parantaen käytettävyyttä. (Hong 2018.)

```
<div class="modal">
 <div class="modal-dialog">
  <div class="modal-content">
   <div class="modal-header">
    <h5 class="modal-title">Modal title</h5>
    <button type="button"
     class="close"
    data-dismiss="modal">
     <span>&times;</span>
    </button>
   </div>
   <div class="modal-body">
    <p>Modal body text goes here.</p>
   </div>
   <div class="modal-footer">
    <button type="button" class="btn btn-primary">Save changes</button>
    <button type="button"
     class="btn btn-secondary"
     data-dismiss="modal">Close</button>
   </div>
  </div>
 </div>
</div>
```

Kuvio 34. Merkintäkoodi vakiomodaalille

Modaalin ulkoasu ja sen rakenne (kuvio 34) ovat hyvin samanlaisia kuin kortin ulkoasu ja rakenne. Esimerkiksi yhteiset komponenttien osat kuten `-body`, `-title`, `-header`, `-footer` ja niin edespäin, ovat etuliitteellä `modal` korteissa käytetyn etuliitteen sijaan (kuvio 32).

Huomattavaa modaalin toiminnassa kuitenkin on, että ne eivät toimi pelkillä HTML- ja CSS-kielillä, vaan vaativat toimiakseen Bootstrap-kirjaston JavaScript-liitännäiskoodit. Esimerkiksi modalin avaaminen ja sulkeminen vaativat JavaScript-logiikkaa. Modaalin sulkemista voidaan kutsua elementin kautta, jolla on `data-dismiss`-attribuutti (kuvio 34).

8 TOTEUTUS

Tässä luvussa esitellään ja määritetään työssä toteutettu edellisten lukujen teoriaa ja tekniikoita soveltava verkkokäyttöliittymä. Sovellus ja sen lähdekoodi ovat avoimesti nähtävillä Githubissa.

8.1 Sovellusmääritys

Toteutettava sovellus on avoimesti saatavilla verkossa. Sovelluksen toteuttamisessa käytetään git-versionhallintajärjestelmää ja se tehdään julkiseksi Github-palvelun avulla. Sovelluksen ulkoasu mallintaa tyypillistä sosiaalisen median sovelluksen verkkoasettelua. Sovelluksen ja sen käyttöliittymän toteuttamisessa käytetään Aurelia- ja Bootstrap-sovelluskehyksiä. Verkkosivu ja sen lähdekoodin versionhallinta ylläpidetään Githubissa. Sovelluksen käyttöliittymään toteutetaan kirjautuminen ja käyttäjän profiilisivu. Sovelluksen ei tule toteuttaa varsinaista autentikointia tai tukea yksilöityjä käyttäjiä, vaan kirjautuminen on simuloitua. Myös käyttäjän profiilisivun sisältö luodaan satunnaisesti. Sovelluksen käyttöliittymän tulee skaalautua mobiililaitteille ja pyötäkonekokoisille laitteille.

8.2 Lähdekoodi ja lopputuote

Sovelluksen lähdekoodin historia ja itse lopputuote on saatavilla ja nähtävissä julkisessa Github-repositoriossa. Verkkosivua isännöidään Github pages -ympäristössä, jolla voi repositorion yhteydessä tarjota staattisia verkkosivuja. Kehittämisen ja julkaisemisen helpottamiseksi projektiin on kytketty automaattinen käännös ja julkaisuprosessi Github actions -toiminnallisuutta hyödyntämällä.

VanhanenJLA Create .yml — Latest commit c0815b7 7 days ago — History

1 contributor

33 lines (28 sloc) | 719 Bytes — Raw | Blame

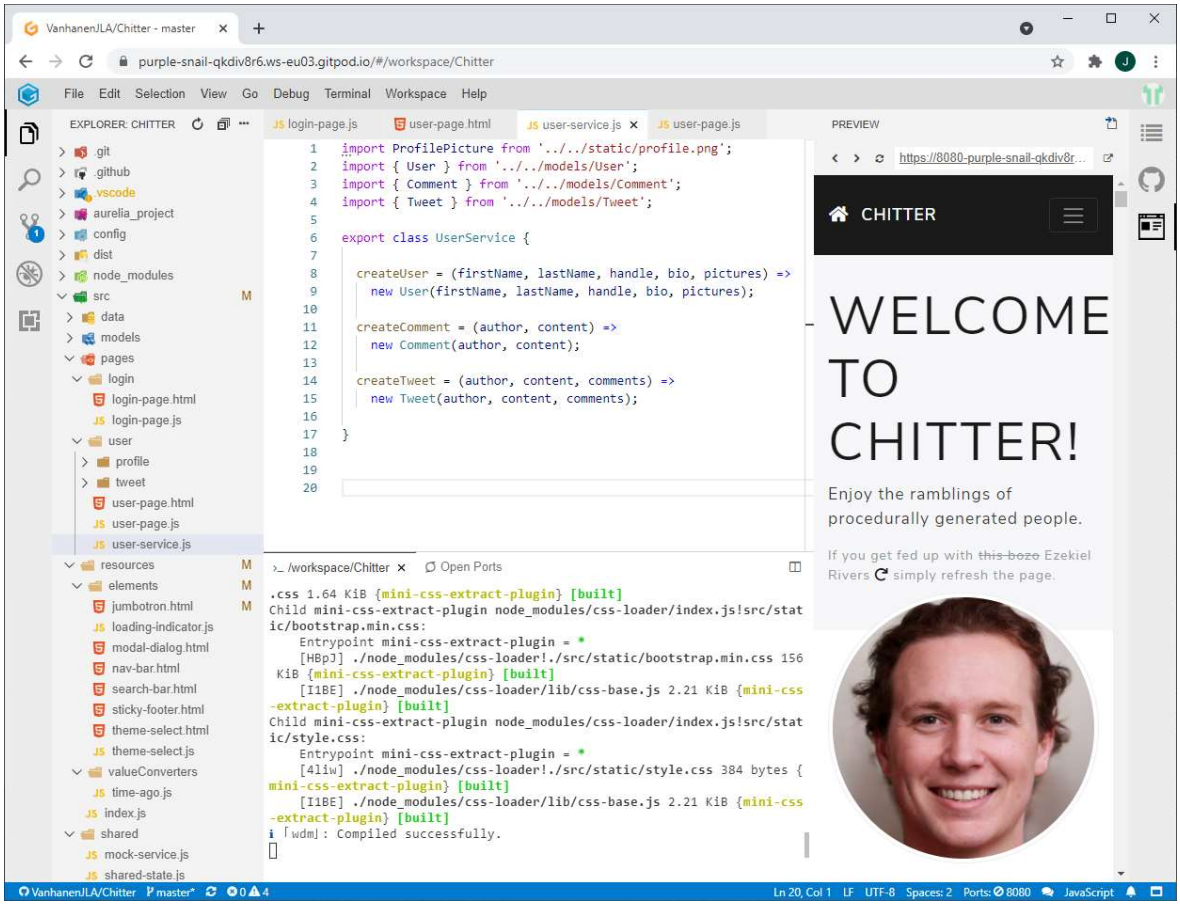
```yaml
name: CD

on:
  push:
    branches: [master]
  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Setup Node.js environment
        uses: actions/setup-node@v2.1.4
        with:
          node-version: "14.x"

      - name: Checkout
        uses: actions/checkout@v2.3.4
        with:
          persist-credentials: false

      - name: Install and Build
        working-directory: ./
        run: |
          npm install
          npm run build
      - name: Deploy to GitHub Pages
        uses: JamesIves/github-pages-deploy-action@3.7.1

        with:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          BRANCH: gh-pages
          FOLDER: ./dist
```

Kuvio 35. Jatkuvan julkaisun (CD) skripti tarkasteltuna Githubin verkkokäyttöliittymästä.

Kuviossa 35 nähdään projektiin määritetyn jatkuvan julkaisun skriptin lähdekoodin tila kirjoitushetkellä. Käyttöliittymä esittää myös tietoja viimeisimmästä muutoksesta tarkasteltavaan tiedostoon, kuten esimerkiksi kommitin otsikon, suorittajan ja suorituksen ajanhetken.

CD-skripti määrittää, että aina projektin versionhallinnan master-haaraan muutoksia puskettaessa ajetaan kuvion 35 riviltä 8 alkava jobi. Jobi kääntää ja julkaiseee pro-

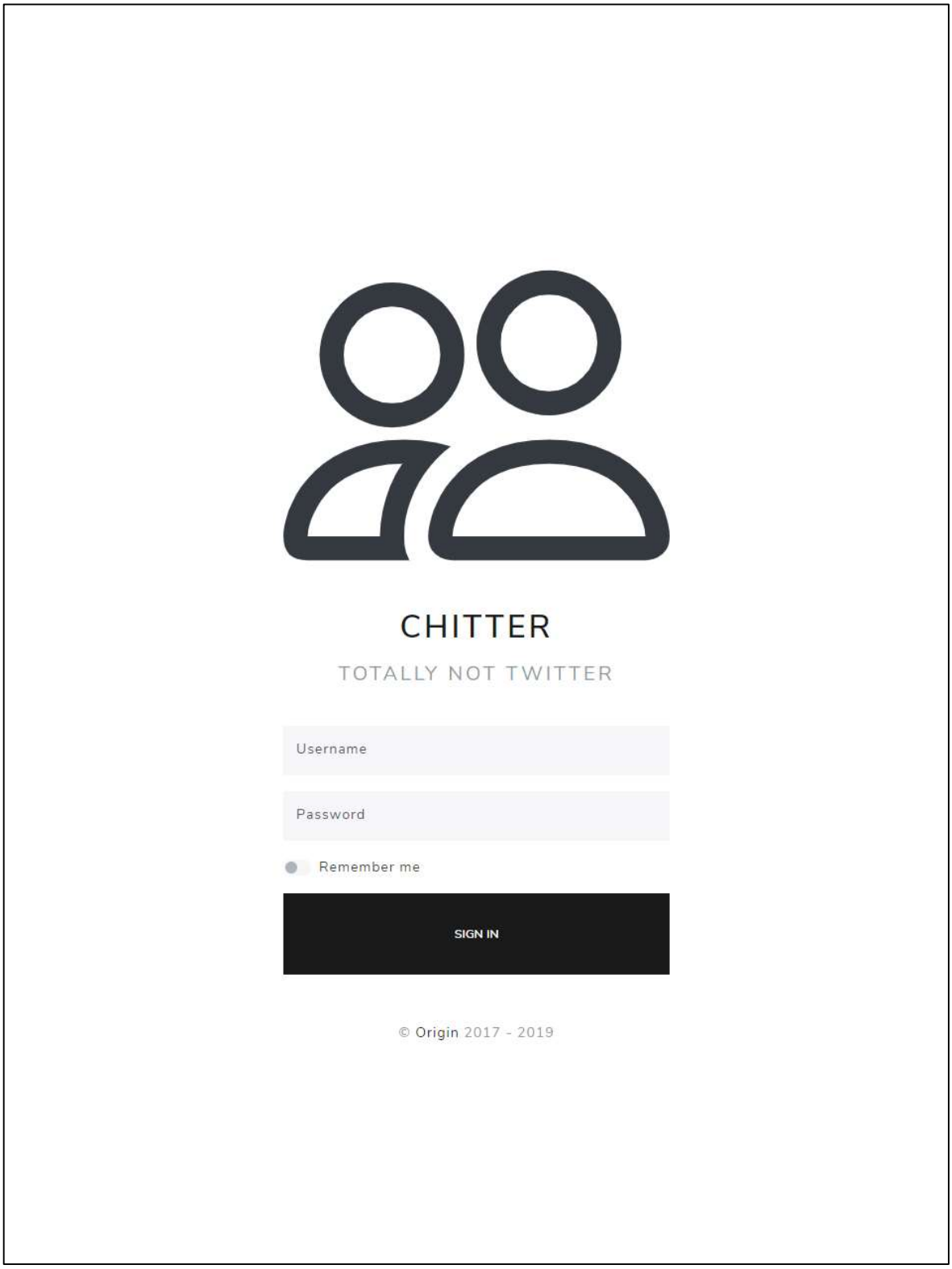
jektista uuden version gh-pages-haaraan. Jobin ajettua päätyvät kommiteissa pusketut viimeisimmät muutokset välittömästi ns. tuotantoon, eli jaettavalla verkkosivulla näkyy sovelluksesta uusin versio.



Kuvio 36. Gitpod-palvelun näkymä projektin verkkokehitysympäristöstä.

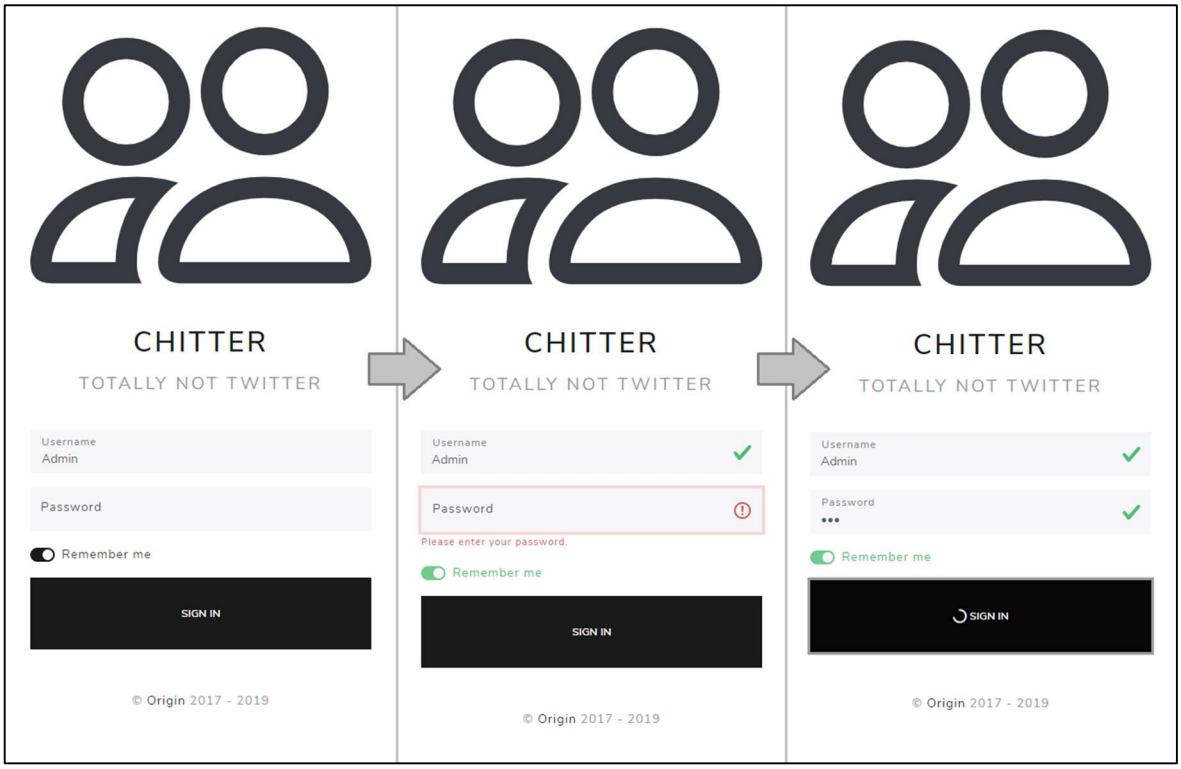
Projektin kehitysympäristön alustaminen on myös automoitu Gitpod-palvelulla (kuvio 36). Projektin kehittäminen on mahdollista pelkällä selaimella, verkko-osoitteella ja verkkoversionhallintapalveluntarjoajan (Github, GitLab, Bitbucket) tunnuksilla. Projektin Gitpod-ympäristön osoitteeseen navigoitaessa Gitpod-palvelu varaa käyttäjälle yksilöllisen virtuaalikoneen, jolle projekti käynnistyy. Instanssin valmistuttua on projektia mahdollista kehittää Visual Studio Coden selainversiolla, joka aukeaa selaimeen.

8.3 Kirjautuminen



Kuvio 37. Kirjautumisnäkymä.

Sovelluksen käyttöliittymään on toteutettu kirjautumisnäkymä (kuvio 37). Kirjautumisnäkymän elementit ovat sarakkeessa, joka on keskitettynä viewporttiin. Kirjautumislomake muodostuu logosta, sovelluksen nimi-otsakkeesta, aliotsikosta ja tunnistautumisen syötekentistä.



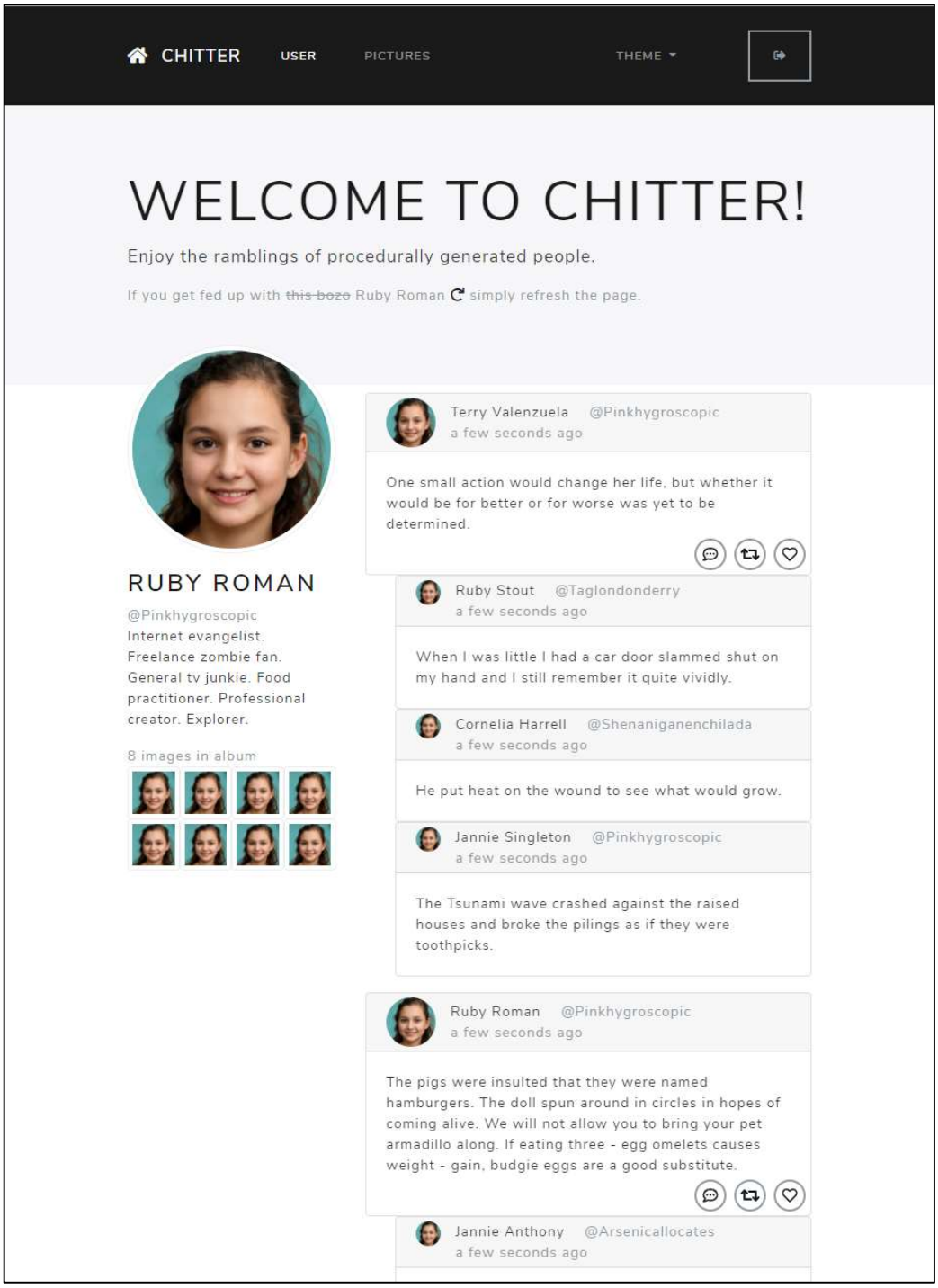
Kuvio 38. Kirjautumisprosessin kulku.

Kuviossa 38 havainnollistetaan lomakkeen ulkoasun mukautumista käyttäjän syöttäessä tunnistautumistaan. Siinä kuvataan kirjautumisprosessi, jossa käyttäjä syöttää ensiksi vaillinaiset tiedot lomakkeeseen ja yrittää kirjautua sisään, jolloin käyttöliittymä viestii käyttäjälle virheellisistä syötteistä. Käyttäjä korjaa virheelliset syötteensä ja käyttöliittymä viestii lopuksi hyväksyvänsä käyttäjän kirjautumisyrityksen.

Nähdään, että kirjautumislomakkeen syötekenttien vihjeet väistyvät käyttäjän kirjoittaessa syötekenttään, jolloin syötekentän tarkoitus on tulkittavissa vielä syöttämisen jälkeenkin. Kun käyttäjä on painanut sisäänkirjautumispainiketta syöttämättä salasanaa nähdään, että käyttöliittymä validoi lomakkeen. Hyväksytyt syötteet värittyvät vihreiksi ja hylätyt punaisiksi. Hylättyjen yhteydessä kerrotaan myös syy hylättyyn validointiin. Esimerkin tapauksessa salasanan syötekenttä on tyhjä ja käyttöliittymä kehoittaa käyttäjää syöttämään salasanansa.

Sovelluksen kirjautumistoiminnallisuus ei autentikoi käyttäjiä oikeasti, vaan ainoastaan tarkastaa, että syötekenttiin on syötetty jotain. Hyväksytyillä tunnuksilla kirjauduttaessa käyttöliittymä myös simuloi sisään kirjautumisen viivettä muutaman sekunnin ajan, jolloin lomakkeen kirjautumisnapissa näytetään latausindikaattoria.

Profiili

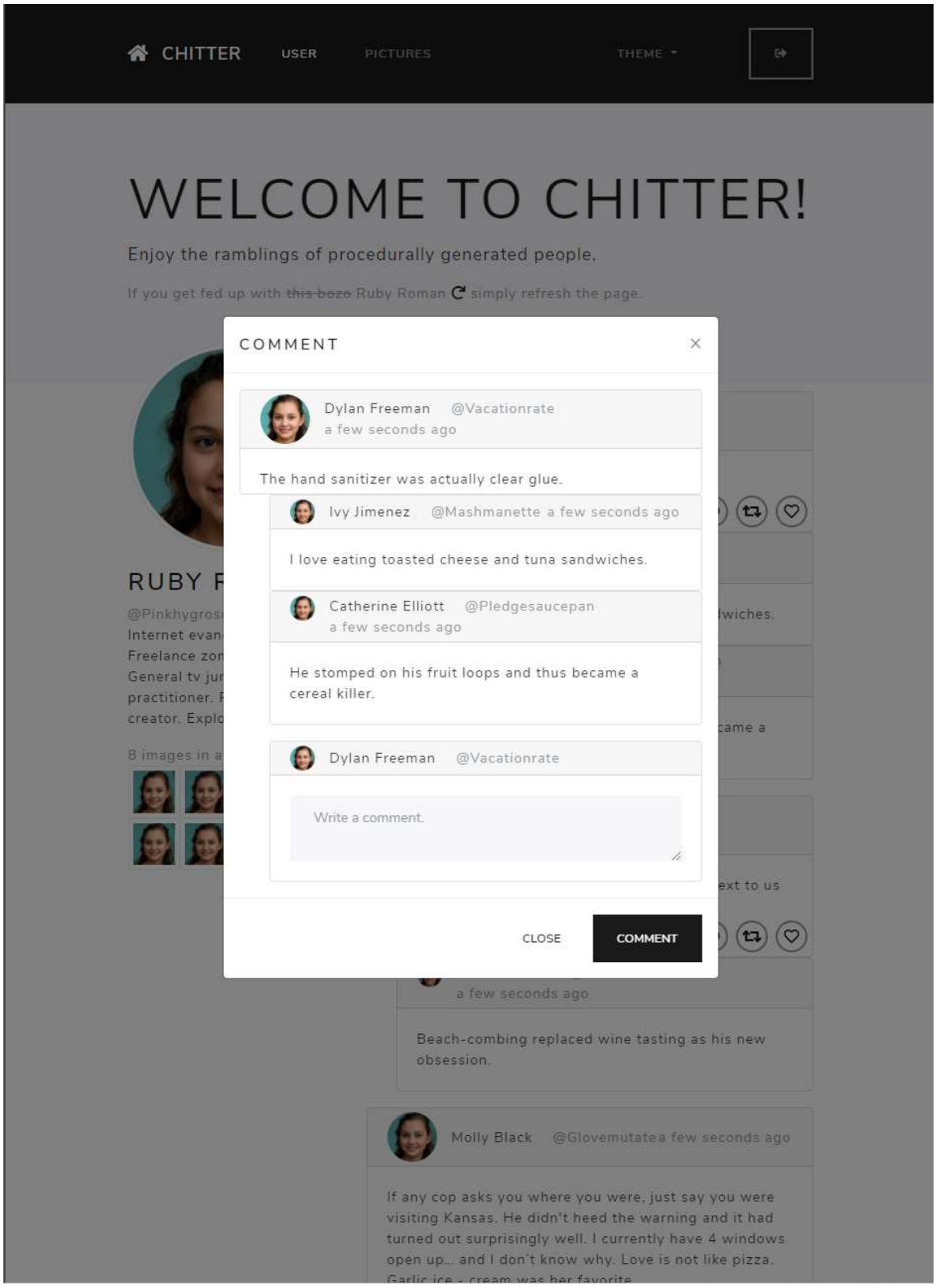


Kuvio 39. Käyttäjän profiilinäkymä.

Onnistuneen kirjautumisen jälkeen aukeaa Chitter-demosovellus satunnaisesti generoidun käyttäjän profiilinäkymään (kuvio 39). Korostettakoon vielä, että sovelluksen tiedot kuten kuvat, nimet tai viestit eivät ole oikeita. Esimerkiksi kuvat sovellukseen haetaan palvelusta, joka tarjoaa tekoälyn luomia ihmisten kasvokuvia.

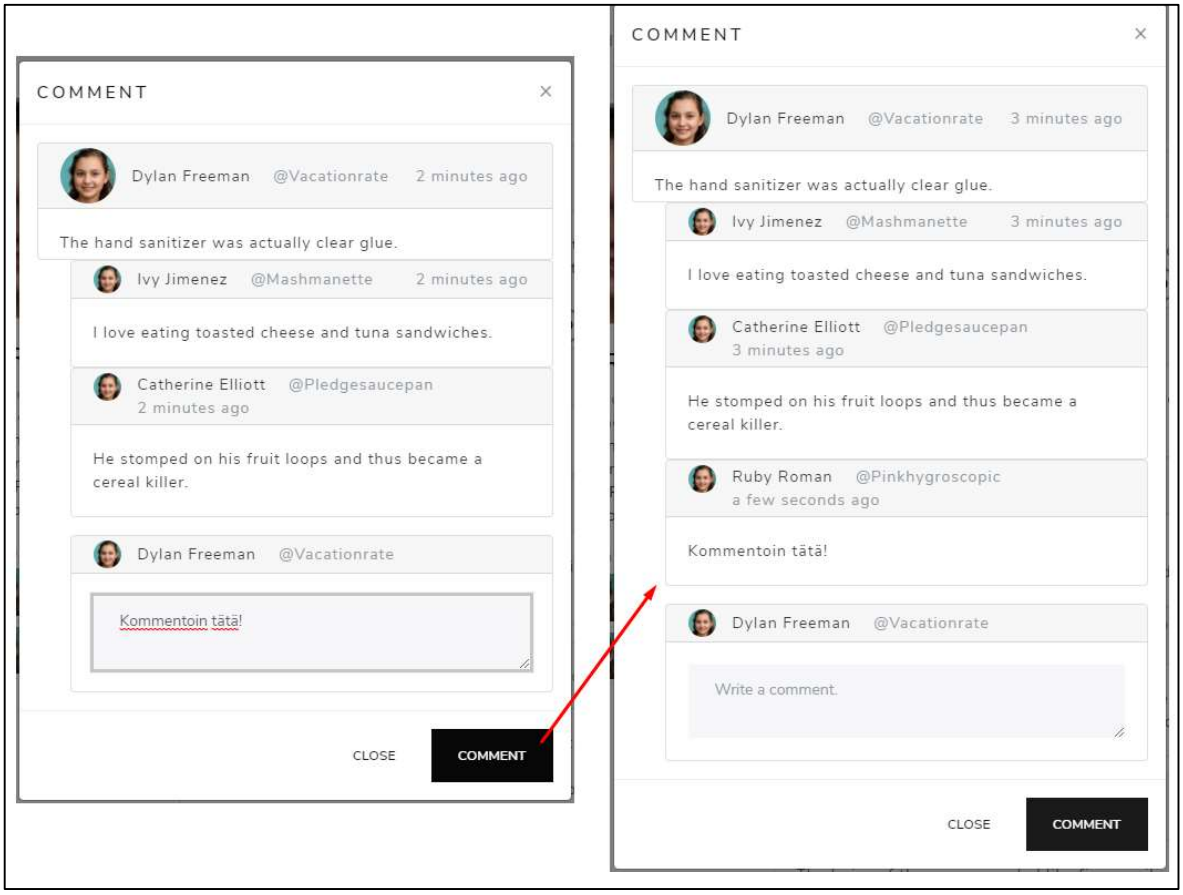
Profiilinäkymä koostuu käyttäjän tietojen sarakkeesta ja viestien aikajanasta. Sovelluksessa käyttäjän tietomalli muodostuu etu- ja sukunimistä, yksilöivästä profiilitunnisteesta, profiilitekstistä ja kokoelmasta kuvia tiedot esitetään käyttöliittymässä profile-details-komponentilla.

Käyttäjän sivulle saapuu näkyviin myös ajastettuna viestejä, joille luodaan kommentteja satunnaisilta henkilöiltä. Viestin ja kommentin erottaa kortin leveydestä ja toisiinsa liittyvät kortit ovat kiinni toisissaan. Käyttöliittymästä voi myös tarkastella käyttäjän profiilikuvaa suurennettuna ponnahdusikkunassa ja tykätä, sekä kommentoida viestejä.



Kuvio 40. Kommentointimodaali.

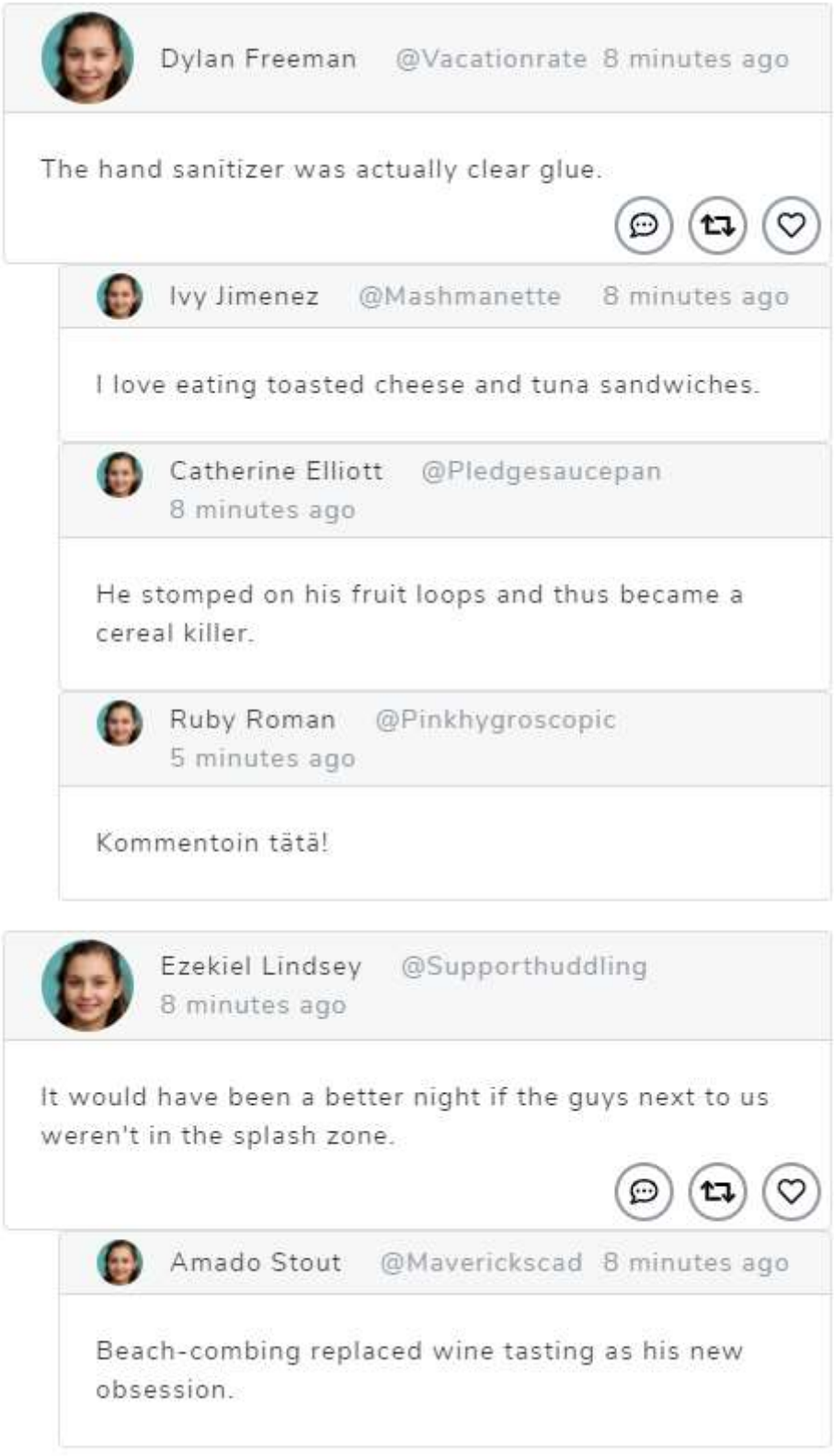
Viesti-komponentin pyöreästä puhekupla-kuvakkeisesta painikkeesta aukeaa modaali-ikkuna viestin kommentoimiseen (kuvio 40 ja kuvio 41).



Kuvio 41. Kommentoiminen modalissa.

Viestiä kommentoitaessa uusi keskeneräinen kommentti on erotettu muista kommenteista pienellä välistyksellä ja sen komponentissa on textarea-syötekenttä (kuvio 41). Kommentti lähetetään painamalla modaalin alatunnisteessa sijaitsevaa kommentoi-painiketta, jolloin lähetetty kommentti tulee näkyviin viestin kommenttien perään.

Tarkkasilmäinen saattaa myös huomata, että kommentin luomiskomponentissa ja luodussa kommentissa on eri käyttäjät. Kommenttia luotaessa esitetään viestin käyttäjää ja kommenttiin päätyykin profiilisivun käyttäjä. Tässä on havaittavissa ohjelmointivirhe, jossa uuden kommentin näkymään on sidottu väärä käyttäjä (kuvio 41).

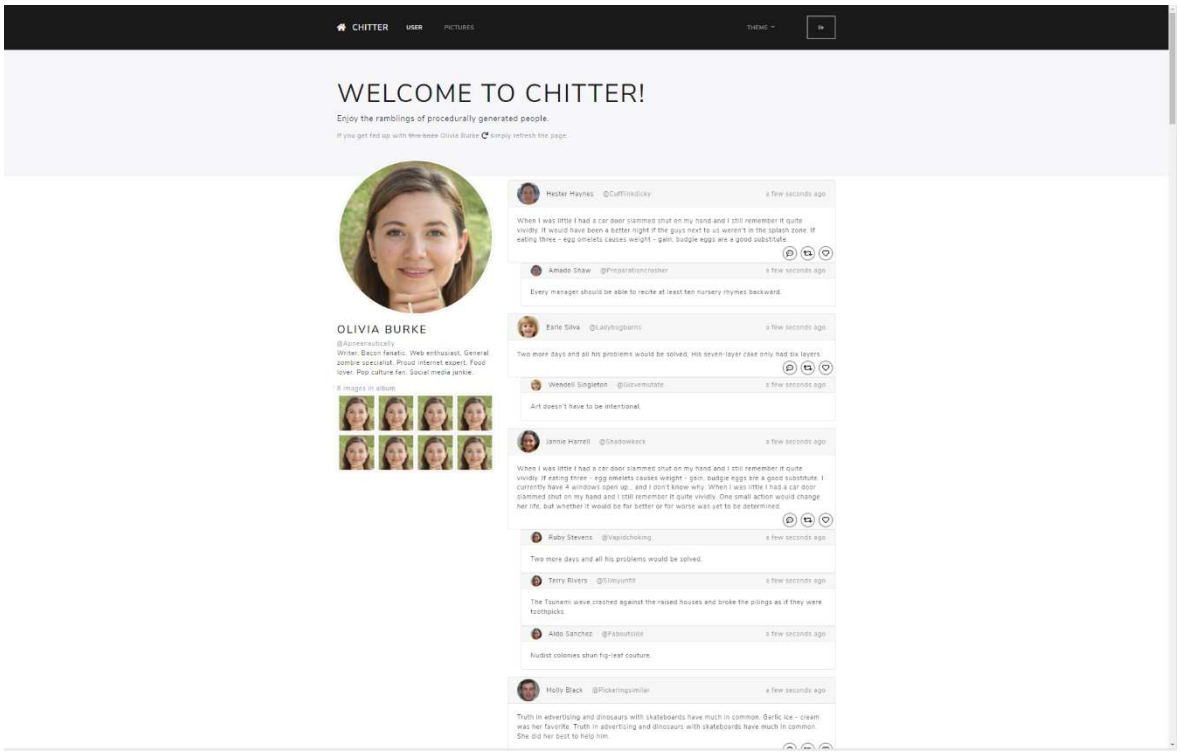


Kuvio 42. Lähetetty kommentti aikajanalla.

Käyttäjän kommentoidessa viestiä päivittyy lähetetty kommentti myös profiilisivun aikajanalle (kuvio 42). Kuvion 40 ”Kommentoin tätä!”-kommentti näkyy siis nyt myös

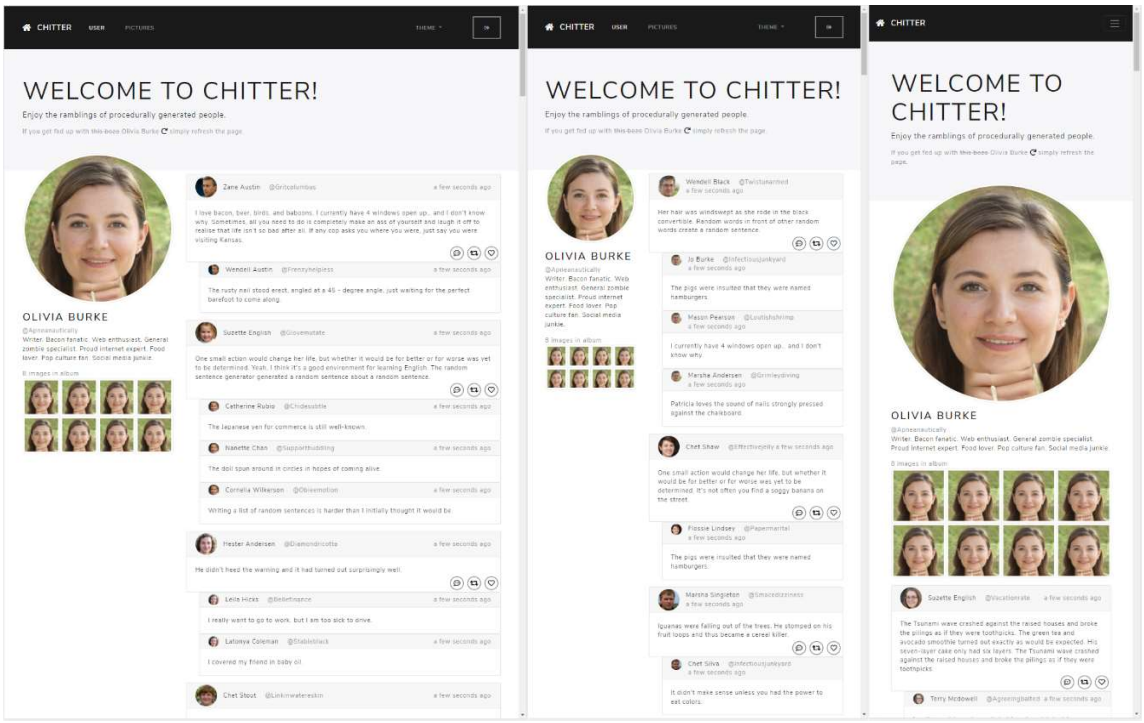
aikajanalla. Viestien ja kommenttien korteissa huomattavaa on niiden otsakkeessa esitetty aikaleima, joka on tyylitelty momentjs-kirjastolla esittämään lähetyksestä kulunut aika käyttäjäystävällisessä muodossa. Aikaleimat myös pysyvät oikeassa ajassa ilman, että sivua tarvitsee ladata uudelleen.

8.4 Responsiivisuus



Kuvio 43. Verkkosivu pöytäkonekoossa tarkasteltuna.

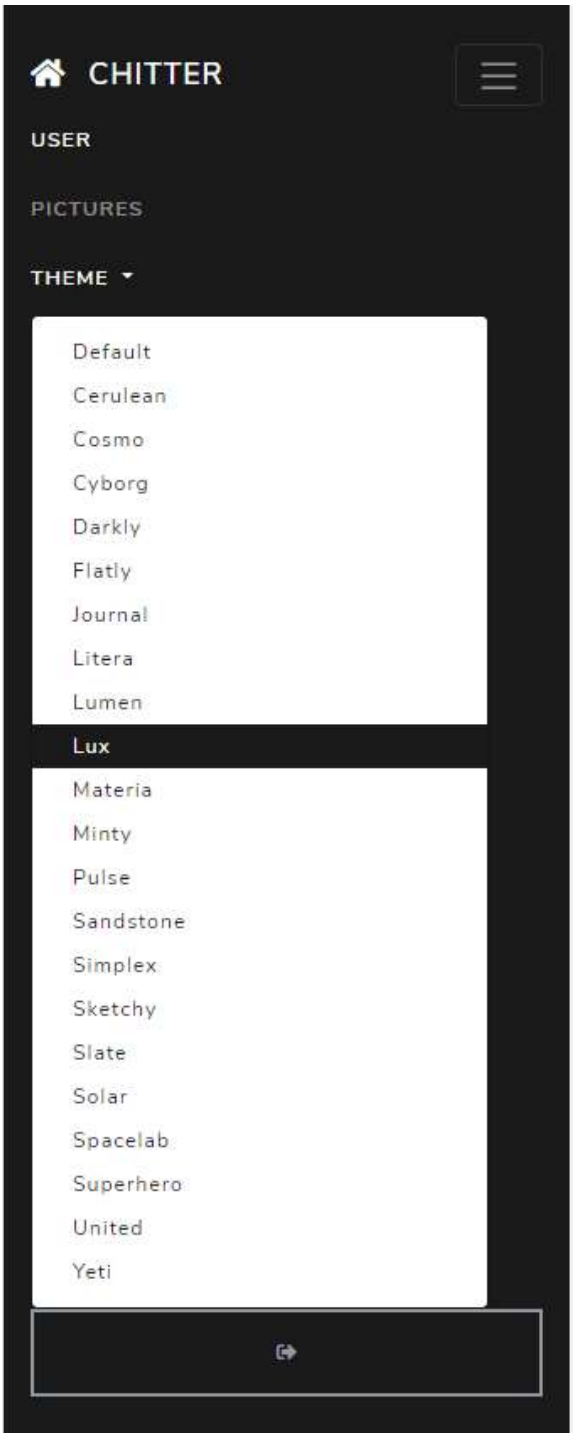
Sovelluksen käyttöliittymä on upotettuna bootstrap-kirjaston container-komponenttiin, jolla rajataan sisältöä venymästä koko ruudun levyiseksi suuremmilla näyttökoilla. Näkymän leveyden rajaaminen parantaa käyttökokemusta huomattavasti esimerkiksi luettavalle sisällölle.



Kuvio 44. Käyttöliittymän skaalautuminen ruudun koon pienentyessä.

Profiilisivun sarakkeet on määritelty bootstrap-kirjaston responsiivisilla pysäytyspisteillä asettumaan allekkain, mikäli päätelaitteen ruudunleveys pienenee alle medium-pysäytyspisteen. Kuviossa 44 nähdään. että esimerkiksi mobiililaitteella tarkasteltaessa käyttöliittymässä asettuvat käyttäjän tiedot ennen aikajanaa. Käyttöliittymä on siis yhtä käyttäjäystävällinen niin pienillä kuin suurillekin ruuduilla.

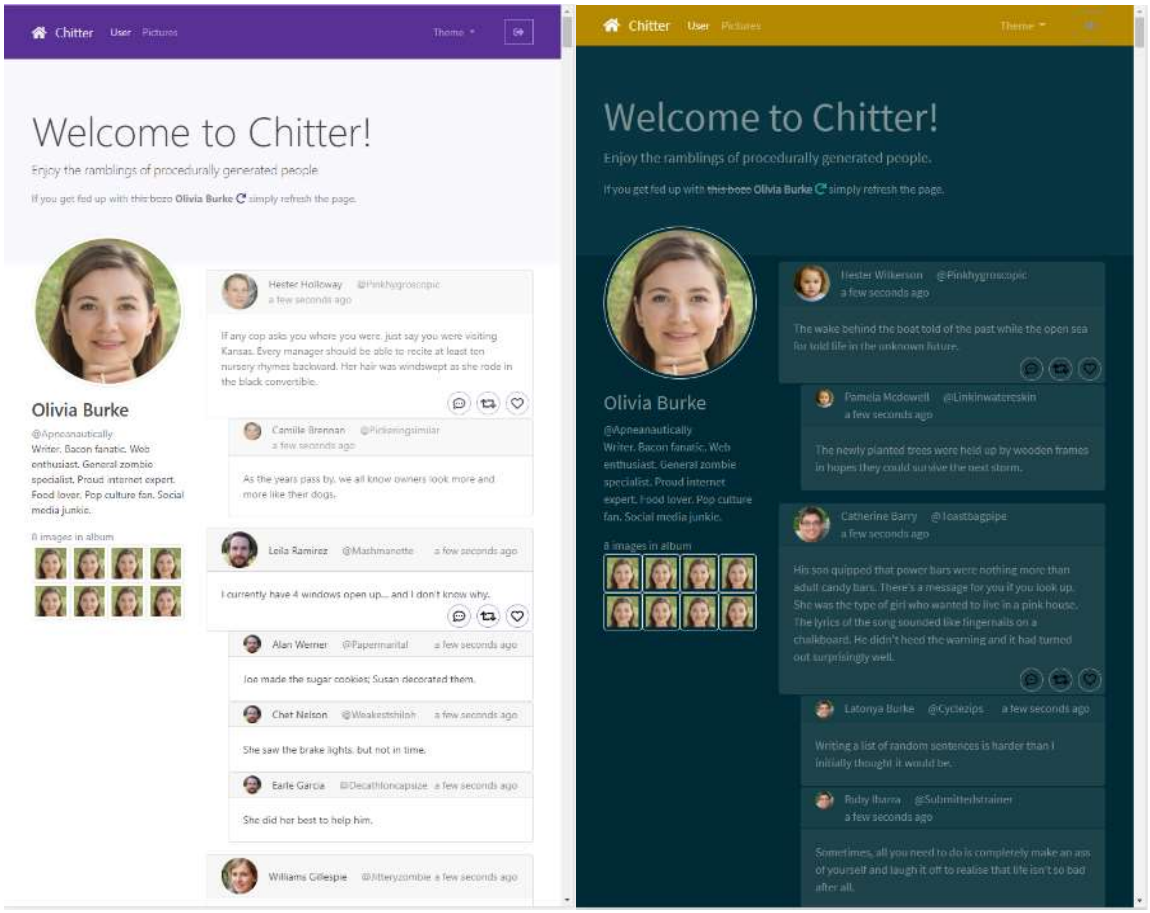
8.5 Ulkoasun teemat



Kuvio 45. Navigaatiopalkin teemavalitsin.

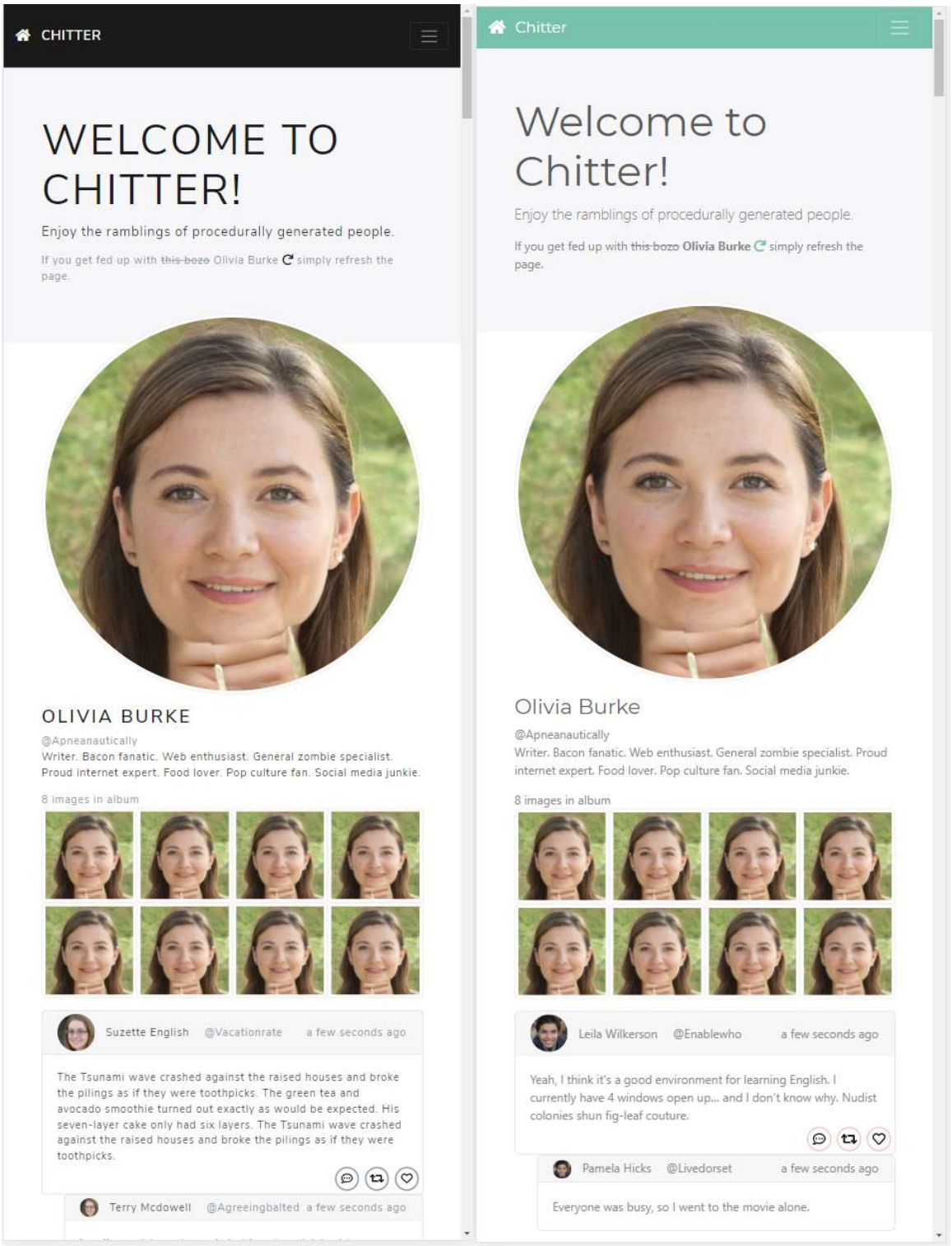
Sovellukseen on myös toteutettu valitsin käyttöliittymän ulkoasun teemalle (kuvio 45). Valitsimen tarjoamat teemat ovat bootstrap-kirjaston vakio css-määreiden pohjalta rakennettuja vaihtoehtoisia tyylejä. Tähän asti esitellyissä kuvissa käyttöliittymässä on ollut valittuna Lux-teema.

Tyylitiedostot on lisätty projektiin sisällönjakeluverkosta viittaamalla, joten teemavalitsin varsinaisesti muuttaa vain voimassa olevaa tyylitiedostoa. Itse teemat eivät siis edellytä ylimääräistä ohjelmoimista, mutta niiden käyttäminen vaatii, että käyttöliittymän tyyli määritetään niiden tukemissa rajoissa. Esimerkiksi ohjelmoijan kovakoodaamat tyylit kuten värit eivät enää muuttuisi teemojen vaihdon yhteydessä.



Kuvio 46. Esimerkki teemoista Pulse ja Solar.

Teemojen väliset erot korostuvat paremmin itse verkkosovelluksessa, mutta kuviosta 46 nähdään ainakin, että teemat vaikuttavat suurimmaksi osaksi kirjaston väripalettiin ja joidenkin komponenttien kuten esimerkiksi navigointipalkin vakiokorkeuteen. Teemoilla on myös mahdollista helposti toteuttaa käyttöliittymän tumma- ja vaalea-moodit, jotka ovat alkaneet yleistyä laitteissa ja palveluissa natiivisti, mutta ainakin näillä bootstrap-teemoilla toinen tiloista kärsii, sillä osalle komponenteista määritellään kirkkauteen tai tummuuteen sidottu tyyliluokka, joka ei ainakaan ilman ylimääräistä ohjelmalogiikkaa vaihdu teemojen mukana.



Kuvio 47. Mobiilikokoiset Lux- ja Minty-teemat.

Kuviosta 47 erottuu hieman paremmin kuin kuviosta 46, kuinka teemat vaikuttavat myös typografiaan. Lux-teemassa otsakkeiden kirjasimet ovat kaikki kapitalisoitu ja tekstin kontrasti on suurempi kuin Minty-teemassa.

9 YHTEENVETO

Opinnäytetyössä käsiteltiin suunnittelumalleja olio-ohjelmoinnin näkökulmasta. Työssä esiteltiin ohjelmistoarkkitehtuuri ja laatutekijät ja korostettiin niiden merkitystä ohjelmoijalle ja ohjelmistotuotannolle. Esiteltiin arkkitehtuurimallit eli suunnittelumallit joiden tarkkuusalue on koko ohjelmisto ja erityisesti kerrosarkkitehtuurimalli. Esitysmalleista käsiteltiin pelkkää esityskerrosta koskettavia MVC-, MVU- ja MVVM-esitysmalleja. Tarkemmin esiteltiin Aurelia-sovelluskehystä. Sen merkittävämpiä ominaisuuksia käsiteltiin hyödyntäen MVVM-esitysmallin periaatteita. Seuraavaksi tutustuttiin Bootstrap-kirjastoon ja käyttöliittymien responsiiviseen toteuttamiseen ja esiteltiin muutamia kirjaston keskeisimmistä komponenteista.

Opinnäytetyössä luotiin yleiskuva miten eritasoiset suunnittelumallit liittyvät toisiinsa. Erityisesti esitysmallien yhteydessä esiteltiin rinnakkaisia ratkaisuja esityskerroksen haasteisiin. Opinnäytetyössä myös toteutettiin käsitellyn teorian pohjalta verkkosovellus ja -käyttöliittymä Aurelia- ja Bootstrap-sovelluskehyksiä hyödyntäen.

Opinnäytetyöhön teetetyn tutkimustyön myötä on ollut huomattavasti helpompaa siirtyä eri verkkotekniikoiden ja niiden tarjoamien ratkaisuiden välillä, sillä tarpeet ja ratkottavat ongelmat tekniikoiden pohjalla ovat pitkälti samat. Myös uusien projektien koodikantoihin tutustuminen on ollut helpompaa, mikäli niissä on noudatettu sovittuja ja yleisiä käytäntöjä tai ratkaisuja.

Mikäli sovellukselle olisi varsinainen käyttökohde ja jatkokehitystarve, olisi siihen helppo palata ja matala kynnys aloittaa esiteltyjen projektin kehitystä helpottavien palvelujen avulla kuten Github ja Gitpod. Opinnäytetyössä toteuttamattomia mahdollisia jatkokehitettäviä ominaisuuksia ohjelmistokehityksen kannalta voisivat olla esimerkiksi käyttöliittymän automaatiotestit, jotka voitaisiin kytkeä ajettaviksi ja edellytykseksi automaattiselle julkaisulle.

Projektiin voisi myös toteuttaa kerrosarkkitehtuurissa esiteltyjä muita kerroksia, kuten tiedonpysyvyyden MongoDB-pilvitietokantapalvelulla tai luoda verkkosovellukselle palvelintoteutuksen, siirtää sovelluslogiikka palvelimelle ja yhdistää kerrokset REST-rajapinnoin.

LÄHTEET

Agile Education Research. 2019. Web-palvelinohjelmointi, Java. [Verkkosivu]. O'Reilly Media, Inc. [Viitattu 1.4.2020]. Saatavana: https://web-palvelinohjelmointi-s19.mooc.fi

Blue Spire. 2020a. Aurelia – Templating: HTML Behaviors Introduction. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://aurelia.io/docs/templating/html-behaviors

Blue Spire. 2020b. What is Aurelia? [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://aurelia.io/docs/overview/what-is-aurelia

Blue Spire. 2020c. Aurelia – Templating: Basics. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://aurelia.io/docs/overview/what-is-aurelia

Blue Spire. 2020d. Aurelia – Templating: Custom Elements Basics. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://aurelia.io/docs/templating/custom-elements

Blue Spire. 2020e. Aurelia – Binding: Binding Behaviors. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://aurelia.io/docs/binding/binding-behaviors

Blue Spire. 2020f. Aurelia – Templating: Binding. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://aurelia.io/docs/templating/basics#binding

Duffield, M. 2018. Practical App Development with Aurelia: Leverage the Power of Aurelia to Build Personal and Business Applications. [Verkkokirja]. Apress. [Viitattu 1.4.2020]. Saatavana O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Ecma International. 2015. ECMAScript® 2015 Language Specification. [Viitattu 1.4.2020].Saatavana: https://262.ecma-international.org/6.0/#sec-template-literals

Elm-lang. 2020. The Elm Architecture. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://guide.elm-lang.org/architecture/

Fairbank, J. 2019. Programming Elm. [Verkkokirja]. Pragmatic Bookshelf. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Google. 2021. Introduction to components and templates. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://angular.io/guide/architecture-components#data-binding

Griffith, M. 2017. Why Elm? [Verkkokirja]. O'Reilly Media, Inc. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Hall, C. 2011. ActionScript Developer's Guide to PureMVC. [Verkkokirja]. Helsingin Yliopisto, verkkokurssi. [Viitattu 1.4.2020]. Saatavana: https://learning.oreilly.com/library/view/actionscript-developers-guide/9781449324698/

Hong, P. 2018. Practical Web Design. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Imsirovic, A. 2018. Elm Web Development. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Ingeno, J. 2018. Software Architect's Handbook. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Johnson, R., Helm, R., Vissiedi, J. & Gamma, E. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. [Verkkokirja]. Addison Wesley Professional. [Viitattu 1.4.2020]. Saatavana O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Keeling, M. 2017. Design It!. [Verkkokirja]. Pragmatic Bookshelf. [Viitattu 1.4.2020]. Saatavana O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Knockoutjs. 2021. Bindings – Component binding. [Verkkosivu]. [Viitattu 1.4.2020]. https://knockoutjs.com/documentation/component-binding.html

Lambert, M., Jobsen, B., Cochran, D. & Whitley, I. 2017. Complete Bootstrap: Responsive Web Development with Bootstrap 4. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Lambert, M. & Marah, J. 2017. Bootstrap 4 – Responsive Web Design. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Marah, J. & Jakobus, B. 2018. Mastering Bootstrap 4 - Second Edition. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Martin, R. C. 2012. The Clean Architecture. [Verkkosivu]. The Clean Code Blog. [Viitattu 1.4.2020]. Saatavana: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Martin, R. C. 2017. Clean Architecture: A Craftsman's Guide to Software Structure and Design, First Edition. [Verkkokirja]. Pearson. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Microsoft. 2017. The Model-View-ViewModel Pattern. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

Microsoft. 2018a. Data binding in depth. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://docs.microsoft.com/en-us/windows/uwp/data-binding/data-binding-in-depth

Microsoft. 2018b. Data binding and MVVM. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://docs.microsoft.com/en-us/windows/uwp/data-binding/data-binding-and-mvvm

Microsoft. 2020. What's new in C# 9.0. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#record-types

Mozilla. 2021a. JavaScript guidelines. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://developer.mozilla.org/en-US/docs/MDN/Guidelines/Code_guidelines/JavaScript

Mozilla. 2021b. Template literals. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Noyes, B. 2006. Data Binding with Windows Forms 2.0: Programming Smart Client Data Applications with .NET. [Verkkokirja]. Addison-Wesley Professional. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Oracle. 2020. Class Record. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Record.html

Poudel, P. 2018. Beginning Elm – Model View Update Part 1. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://elmprogramming.com/model-view-update-part-1.html

Raman, A., Subramanian, H. & Raj, P. 2017. Architectural Patterns. [Verkkokirja]. Addison Wesley Professional. [Viitattu 1.4.2020]. Saatavana O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Richards, M. 2015. Software Architecture Patterns. [Verkkokirja]. O'Reilly Media, Inc. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Sifuentes, E. H. M. & Rojas, D. J. A. 2018. Hands-On Full Stack Web Development with Aurelia. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Tutorials Point. 2021. Aurelia – Component Lifecycle. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://www.tutorialspoint.com/aurelia/aurelia_component_lifecycle.htm

Twitter. 2018a. Boostrap – Components: Buttons. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://getbootstrap.com/docs/4.6/components/buttons/

Twitter. 2018b. Boostrap – Layout: Grid. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://getbootstrap.com/docs/4.6/layout/grid/

Twitter. 2018c. Bootstrap – Components: Cards. [Verkkosivu]. [Viitattu 1.4.2020]. Saatavana: https://getbootstrap.com/docs/4.6/components/card/

Vernon, V. 2016. Domain-Driven Design Distilled. [Verkkokirja]. Addison-Wesley Professional. [Viitattu 1.4.2020]. Saatavana O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Vice, R. & Siddiqi, M. S. 2012. MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF, First Edition. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

Warin, G. 2015. Mastering Spring MVC 4. [Verkkokirja]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana: O'Reilly-verkkokirjasto. Vaatii tunnistautumisen.

WHATWG 2021. HTML Living Standard. [Verkkosivu]. Packt Publishing. [Viitattu 1.4.2020]. Saatavana https://html.spec.whatwg.org/multipage/syntax.html.