KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Tatu Putto

PILVIPOHJAISEN KOODILEIKKEIDEN HALLINTASOVELLUKSEN RAKENTAMINEN

Opinnäytetyö
Helmikuu 2017

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Helmikuu 2017
Tietojenkäsittelyn koulutusohjelma

Tikkarinne 9
80200 JOENSUU
013 260 600

Tekijä(t)
Tatu Putto

Nimeke
Pilvipohjaisen koodileikkeiden hallintasovelluksen rakentaminen

Toimeksiantaja
-

Tiivistelmä

Opinnäytetyön tavoitteena oli toteuttaa pilvipohjainen verkkosovellus koodileikkeiden hallintaan. Verkkosovelluksen yhteyteen ei toteutettu erillistä tietokantaa, vaan datan persistointi ulkoistettiin pilveen GitHubin Gist-palveluun. Toteutuksen taustalla on halu toteuttaa vaihtoehtoista tiedon persistointimetodia hyödyntävä verkkosovellus ja tutustua moderneihin verkkosovelluskehityksen tekniikoihin.

Toiminnallisessa osuudessa toteutettiin selainsovellus, jonka kautta käyttäjä voi syöttää, muokata ja hakea koodileikkeitä Gist-palvelusta. Selainsovellus rakennettiin Redux- ja SPA-arkkitehtuurien mukaisesti React-käyttöliittymäkirjaston sekä React Router-reitityskirjaston avulla. Selainsovelluksen lisäksi rakennettiin yksinkertainen Java-palvelinsovellus, jotta käyttäjä voi auktorisoida selainsovelluksen käyttämään Gist-palvelua käyttäjän nimissä. Raportissa käsitellään selainsovelluksen toteutuksessa käytettyjä työkaluja ja menetelmiä sekä esitellään selainsovelluksen toteutus. Raportissa ei käsitellä palvelinsovelluksen toteutusta.

Opinnäytetyön tuloksena saatiin toimiva verkkosovellus, joka käyttää tehokkaasti Gist-palvelua tiedon persistointiin sekä hyödyntää Redux- ja SPA-arkkitehtuureja. Tuotoksena syntynyttä verkkosovellusta on helppo lähteä kehittämään kohti julkaisuvalmiutta.

Kieli
suomi

Sivuja 67
Liitteet 0
Liitesivumäärä 0

Asiasanat
verkkosovellus, koodileike, käyttöliittymä, React, Redux, yhden sivun sovellus

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
February 2017
Business Information Technology

Tikkarinne 9
FI 80200 JOENSUU
FINLAND
013 260 600

Author(s)
Tatu Putto

Title
Building a Cloud-Based Web Application for Code Snippet Management

Commissioned by
-

Abstract

The aim of this thesis was to implement a cloud-based web application for code snippet management. Instead of using a traditional database, data persistence is outsourced to GitHub's Gist service. The subject for this thesis arose from the author's desire to develop an application which utilizes an alternative way of handling data persistence. Author also wanted to become acquainted with modern web development techniques.

In the practical part of this thesis, a web application was implemented for creating, modifying and fetching code snippets from Gist service. The web application was built with React and React Router JavaScript libraries to support Redux and single-page application architectures. Small Java server application was also created to allow the user to authorize web application to use Gist service on user's behalf. The theoretical part of this thesis familiarizes the reader with the basics of the tools and methods used in the development process of the web application. In addition, implementation details and the resulting web application are examined and presented. The thesis does not cover server application implementation.

The result of this thesis was a fully functional single-page application for code snippet management. The implemented single-page application utilizes cloud-based data persistence model and Redux effectively. In addition the implementation provides a great base for further development.

Language	Pages 67
Finnish	Appendices 0
	Pages of Appendices 0

Keywords
web application, code snippet, user interface, React, Redux, single-page application

Sisältö

Lyhenteet ja termit

API	Application Programming Interface, ohjelmointirajapinta. Joukko määritelmiä, joiden avulla ohjelmat voivat kommunikoida ja vaihtaa tietoa keskenään. (Wikipedia 2015.)
CRUD	Create, read, update and delete. Datan persistoinnin neljä perusfunktiota. (Wikipedia 2017a.)
DOM	Document Object Model, dokumenttioliomalli. DOM on rajapinta, jonka avulla dokumentin sisältöä voidaan lukea ja manipuloida dynaamisesti esimerkiksi JavaScriptin avulla. (W3C 2005.)
ES6	ECMAScript 2015. ES6 on JavaScript-ohjelmointikielen standardi, joka lisää JavaScriptin syntaksiin uusia ominaisuuksia, kuten luokat ja moduulit. (Wikipedia 2017b.)
Git	Hajautettu versionhallintajärjestelmä ohjelmistokehitykseen (Git 2017).
GitHub	Git-versionhallintajärjestelmää hyödyntävä verkkopalvelu, joka tarjoaa mm. säilytyspaikan ohjelmistokehitysprojekteille (Wikipedia 2016a).
JSON	JavaScript Object Notation. Tekstipohjainen tiedostomuoto datan välitykseen. (Wikipedia 2017c.)
Promise	JavaScript-olio asynkronisten toiminnallisuuksien suorittamiseen. Promise-olio edustaa arvoa, joka voi olla saatavilla nyt, tulevaisuudessa tai ei koskaan. (Mozilla 2017.)

REST ”REST (Representational State Transfer) on HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen” (Wikipedia 2016b).

1 Johdanto

Opinnäytetyön tarkoituksena oli toteuttaa pilvipohjainen verkkosovellus koodileikkeiden hallintaan. Pilvipohjaisuudella viitataan tämän opinnäytetyön yhteydessä datan persistoinnin toteutustapaan, joka perinteisen tietokantaratkaisun sijaan ulkoistetaan kokonaan pilveen GitHubin palvelimille. Sovellus toimii vaihtoehtoisena käyttöliittymänä GitHubin Gist-palvelulle. Sovellus kommunikoi GitHubin palvelimien kanssa GitHubin tarjoamien API:en välityksellä (Gist ja Authorization API).

Opinnäytetyön aihe on oma ideani. Valitsin tämän idean opinnäytetyöni aiheeksi sen sopivan laajuuden sekä mielenkiintoisten teknisten haasteiden takia. Koska toteutus on hyvinkin joustava, haastavuus voidaan asettaa sellaiselle tasolle, että työn lopputulos tuo esille osaamistani web-kehittäjänä ja mahdollistaa ammatillisen kehittymisen projektin aikana. Lisäksi olen jo pitkään halunnut toteuttaa verkkosovelluksen, joka hyödyntää SPA-arkkitehtuuria ja käyttää vaihtoehtoista tiedon persistointimetodia perinteisen tietokantaratkaisun sijaan.

Työn pääpaino on toiminnallisessa osuudessa, raportissa keskitytään avaamaan toiminnallisen osuuden toteutusta. Raportissa esitellään pinnallisella tasolla keskeisimmät projektin aikana käytetyt työkalut ja menetelmät.

Opinnäytetyön ensisijaisena tavoitteena on rakentaa koodileikkeiden hallintaan toimiva verkkosovellus, joka hyödyntää tehokkaasti pilvipohjaista tiedon persistointimetodia ja Redux- sekä SPA-arkkitehtuureita. Sovelluksen tulee sisältää käyttäjän koodileikkeiden hallintaan seuraavat toiminnot: luominen, lukeminen, muokkaaminen, poistaminen ja suosikiksi asettaminen. Käyttäjän omien koodileikkeiden käsittelyn lisäksi sovellukseen toteutetaan ns. discover-toiminto. Discover-toiminnon kautta käyttäjä voi tarkastella muiden käyttäjien Gist-palveluun lisäämiä koodileikkeitä. Käyttäjän tulee myös pystyä kopioimaan sekä lisäämään suosikkeihin discover-toiminnon kautta löytämiään koodileikkeitä. Toissijaisena tavoitteena on parantaa Gist-palvelun käytettävyyttä.

2 Kehitysympäristö

2.1 Gist-palvelu

Gist on GitHubin tarjoama palvelu koodileikkeiden hallintaan ja jakamiseen. Gistillä tarkoitetaan Gist-palvelun yhteydessä entiteettiä, joka sisältää yhden tai useamman koodileikkeen sekä muuta kyseiseen kokonaisuuteen liittyvää dataa, kuten tunnistenumeron ja kuvauksen. Gist-palvelu hyödyntää Git-versionhallintasovellusta, joten CRUD-operaatioiden ohella gisteihin voidaan soveltaa versionhallintaa sekä muita GitHubin tarjoamia toiminnallisuuksia, kuten muiden käyttäjien gistien kopiointi omalle GitHub-tilille (fork) ja gistien suosikeiksi asettaminen. (GitHub 2016a.)

Gist-palvelua voidaan käyttää ulkoisesta sovelluksesta GitHubin tarjoaman REST-rajapinnan välityksellä (GitHub 2016b). Tätä mahdollisuutta hyödynnettiin toiminnallisessa osuudessa. Toiminnallisessa osuudessa luotavan sovelluksen yhteyteen ei toteuteta erillistä tietokantaa, vaan datan persistointiin käytetään Gist-palvelua.

Kommunikointi Gist API:n kanssa tapahtuu HTTP-protokollan välityksellä ja datan välitykseen käytetään JSON:ia. API:a käytetään lähettämällä pyyntöjä osoitteeseen https://api.github.com. Osoitteen polkunimi määrittää käytettävän päätepisteen (engl. endpoint), joka puolestaan määrittää minkä resurssin kyseinen pyyntö Gist-palvelusta palauttaa. Esimerkiksi pyyntö osoitteeseen https://api.github.com/gists/public palauttaa uusimmat palveluun luodut julkiset gistit. Sama päätepiste voi toimia useassa eri tehtävässä, riippuen vastaanotetun pyynnön tyypistä. Esimerkiksi /gists-päätepiste voi palauttaa gistejä (GET-tyyppinen pyyntö) tai lisätä gistin palveluun (POST-tyyppinen pyyntö). (GitHub 2016b.)

Gist-palvelua voidaan käyttää API:n välityksellä anonyymisti. Käytännön syistä käyttäjää kuitenkin vaaditaan antamaan sovellukselle valtuudet Gist-palvelun

käyttämiseen käyttäjän GitHub-tilin nimissä. Valtuutus suoritetaan GitHubin Oauth2-valtuutusprotokolla implementaation avulla. Käyttäjälle generoidaan käyttöavain (engl. access token), joka liitetään API:lle lähetettäviin pyyntöihin Authorization-otsikon arvona. Avaimen perusteella API suorittaa toiminnot avaimeen sidotun GitHub-tilin nimissä. (GitHub 2016c.)

2.2 Redux

Yksi sovelluksen toteutuksen suurimmista haasteista oli sovelluksen datan hallitseminen. Data on luonteeltaan dynaamista ja se haetaan sovellukseen asynkronisesti lähes poikkeuksetta. Datan hallitsemiseen tarvittiin selkeä toimintamalli. Varteenotettavia vaihtoehtoja olivat Flux, Redux ja Relay, joista Redux-arkkitehtuurimallin katsottiin vastaavan parhaiten sovelluksen tarpeisiin. Redux pyrkii tekemään sovelluksen datan hallitsemisesta yksinkertaista ja ennalta-arvattavaa rajoittamalla sitä, miten ja milloin dataa voidaan päivittää. Reduxin yhteydessä dataa, jota sovelluksen on kulloinkin hallittava, kutsutaan sovelluksen tilaksi. (Redux 2016a; Redux 2016b.)

2.2.1 Reduxin toimijat

Redux-kokonaisuus koostuu neljästä toimijasta. Jokaisella toimijalla on oma keskeinen roolinsa Reduxin toiminnassa. Reduxin toimijat ovat toiminnot (engl. actions), toiminnon muodostajat (engl. action creators), käsittelijät (engl. reducers) ja varasto (engl. store). (Kubacak 2016.)

Redux-sovelluksessa tilaa voidaan muokata vain lähettämällä toiminto. Toiminto on yksinkertainen JavaScript-olio, joka kuvaa varastolle mitä sovelluksessa tapahtui. Toiminto voidaan ajatella lyhyenä uutisena, esimerkiksi: "Gistin dde8c… hakeminen aloitettiin" (kuva 1). Toiminnon rakenne ja tietosisältö vaihtelee toimintokohtaisesti. Jokaisella toiminnallisuudella on kuitenkin oltava type-ominaisuus suoritettavan toiminnon tyypin tunnistamiseksi. (Redux 2016c; Redux 2016d.)

```
{
    type: 'REQUEST_SELECTED_GIST',
    gistId: 'dde8c74b0c99422c0cc0f6916de22765',
    isFetching: true
}
```

Kuva 1. Gistin haun alkamisesta ilmoittava toiminto.

Toiminto voidaan muodostaa ja lähettää varastolle suoraan käyttöliittymäkomponentista, vastauksena käyttäjän syötteeseen (kuva 2). Toimintojen muodostamista näkymäkerroksessa pyritään kuitenkin välttämään, koska se tekee käyttöliittymäkomponenttien sekä toimintojen uudelleenkäyttämisestä ja testaamisesta vaikeaa. Yleisenä käytäntönä on siirtää toimintojen muodostaminen näkymäkerroksesta ns. toiminnon muodostaja-funktioihin (kuva 3). (Redux 2016d.)

```
<li id={id} onClick={dispatch({type: 'REQUEST_SELECTED_GIST', gistId: id, isFetching: true})}>
```

Kuva 2. Toiminnon muodostaminen ja lähettäminen käyttöliittymäkomponentista.

```
//@EsimerkkiKomponentti.js
<li id={id} onClick={dispatch(requestSelectedGist(id))}>

//@actions.js
function requestSelectedGist(id) {
    return {
        type: 'REQUEST_SELECTED_GIST',
        gistId: 'dde8c74b0c99422c0cc0f6916de22765',
        isFetching: true
    };
}
```

Kuva 3. Toiminnon muodostamisen siirtäminen toiminnon muodostaja-funktiolle.

Toiminnot kuvaavat mitä sovelluksessa tapahtui, mutta ne eivät määrittele miten sovelluksen tila muuttuu vastauksena tapahtumaan, se on käsittelijä-funktioiden tehtävä. Käsittelijä saa argumentteinaan sovelluksen nykyisen tilan sekä toiminnon, joiden perusteella käsittelijä laskee ja palauttaa sovelluksen uuden tilan. Käsittelijän tulee aina olla puhdas funktio. Käsittelijän tulee aina palauttaa

sama arvo, kun sitä kutsutaan samoilla argumenteilla. Käsittelijä ei saa myöskään muokata sille välitettyjä argumentteja tai suorittaa sivuvaikutuksia aiheuttavia toimenpiteitä. Kun käsittelijä-funktiot noudattavat edellä mainittuja sääntöjä, sovelluksen tila muuttuu aina odotetulla tavalla, vastauksena vastaanotettuun toimintoon. (Redux 2016e.)

Käsittelijä-funktio muodostuu switch-valintalausekkeesta (kuva 4). Valintalausekkeena käytetään käsittelijälle välitetyn toiminnon tyyppiä (type-ominaisuuden arvo), jota vastaava tapaus suoritetaan. Jos toiminnon tyyppi ei vastaa yhtäkään valintavakiota, käsittelijä palauttaa vakioarvonsa tai sovelluksen nykyisen tilan. (Redux 2016e.)

```
//Käsittelijä-funktio aktiivisen gistin hallintaan.
export function activeGist(state = {
    gist: {},
    gistId: null,
    isStarred: null,
    isFetching: false,
    fetchError: null
}, action) {
    switch(action.type) {
        //Mitätöidään aktiivinen gist.
        case 'INVALIDATE_GIST':
            return {...state, gist: {}, gistId: null};
            break;
        //Aktiivisen gistin hakeminen aloitettiin.
        case 'REQUEST_SELECTED_GIST':
            return {...state, gistId: action.gistId, isFetching: true};
            break;
        //Aktiivisen gistin hakeminen onnistui.
        case 'RECEIVE_SELECTED_GIST':
            return {...state, gist: action.activeGist, isFetching: false};
            break;
        //Aktiivisen gistin hakeminen onnistui.
        case 'GIST_FETCH_FAILED':
            return {...state, gist: {}, isFetching: false, fetchError: action.error};
            break;

        ...

        //Palautetaan vakioarvot tai nykyinen tila,
        //jos lähetetyn toiminnon tyyppi ei vastaa valintavakioita.
        default:
            return state;
    }
}
```

Kuva 4. Käsittelijä-funktio.

Sovelluksella voi olla yksi tai useampi käsittelijä-funktio. Sovelluksen kasvaessa saavutaan nopeasti tilanteeseen, jossa koko tilan hallitseminen yhdellä käsittelijällä ei ole enää järkevää. Tilan käsittely voidaan jakaa useamman käsittelijän välille pilkkomalla tila osiin. Jokaista tilan osaa kohden luodaan käsittelijä, joka sisältää toiminnallisuudet juuri sen tilan osan hallitsemiseen. Tilan käsittelyn jakamista useamman käsittelijän välille kutsutaan käsittelijä-kompositioksi (engl. reducer composition). Varastolle voidaan määrittää vain yksi käsittelijä. Jotta käsittelijä-kompositiota pystytään hyödyntämään, varastolle määritellään ns, *juurikäsittelijä* (engl. root reducer) (kuva 5). Juurikäsittelijä jakaa tilan osiin, allokoi osat käsittelijöille, koostaa käsittelijöiden paluuarvot yhdeksi olioksi ja lähettää koostetun olion paluuarvonaan varastolle. (Redux 2016e.)

```
function rootReducer(state = {}, action) {
    return {
        activeGist: activeGist(state.activeGist, action),
        filters: filters(state.filters, action),
        gists: gists(state.gists, action),
        notifications: notifications(state.notifications, action),
        pagination: pagination(state.pagination, action),
        user: user(state.user, action)
    };
}
```

Kuva 5. Juurikäsittelijä.

Redux-kokonaisuuden viimeinen osa on varasto. Reduxin toiminta rakentuu varaston ympärille. Varasto käyttää toimintoja ja käsittelijöitä yhteistoiminnassa tilan hallintaan. Varaston pääasiallisena tehtävänä on sovelluksen tilan säilyttäminen. Redux-sovelluksessa varastoja luodaan vain yksi, joten yksi varasto on vastuussa koko sovelluksen tilan säilyttämisestä. Tila kuvataan varastossa oliopuuna (engl. object tree), jota varasto päivittää juurikäsittelijän paluuarvon perusteella. Tilan säilyttämisen ohella varaston tehtäviin kuuluu:

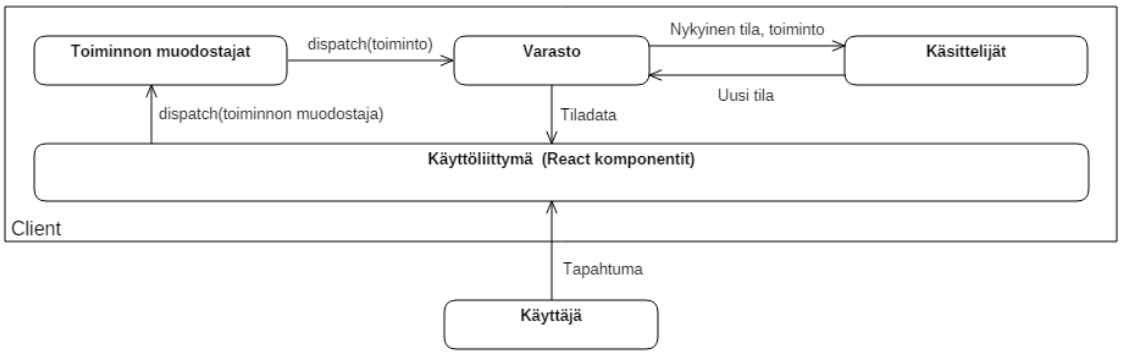
- toiminnon lähettäminen dispatch-metodin välityksellä
- tilan lukemisen mahdollistaminen getState-metodilla
- tapahtumakuuntelijoiden rekisteröiminen subscribe-metodin avulla

- rekisteröityjen tapahtumakuuntelijoiden kutsuminen sovelluksen tilan muuttuessa. (Redux 2016f.)

2.2.2 Tiedonkulku

Kuva 6 esittää tiedonkulkua sovelluksessa. Redux-sovelluksessa tiedonkulku on aina yksisuuntaista. Kun sovelluksen tilaa halutaan muuttaa, muutoksen täytyy aina tapahtua kuvan 6 mukaisesti. Tiedonkulku voidaan jakaa karkeasti viiteen vaiheeseen:

1. Toiminnon muodostajaa kutsutaan dispatch()-funktion välityksellä. Toiminnon muodostaja palauttaa toiminnon, joka välitetään varastolle.
2. Varasto kutsuu käsittelijää ja välittää argumentteina sovelluksen nykyisen tilan ja toiminnon.
3. Käsittelijä laskee sovelluksen uuden tilan välitettyjen argumenttien pohjalta ja lähettää uuden tilan paluuarvona varastolle.
4. Varasto asettaa käsittelijän paluuarvon sovelluksen uudeksi tilaksi ja kutsuu rekisteröityjä komponentteja.
5. Kutsutut komponentit lukevat päivitetyn tilan getState-metodin avulla ja päivittävät näkymän vastaamaan uusia arvoja. (Redux 2016g.)



Kuva 6. Tiedonkulku sovelluksessa.

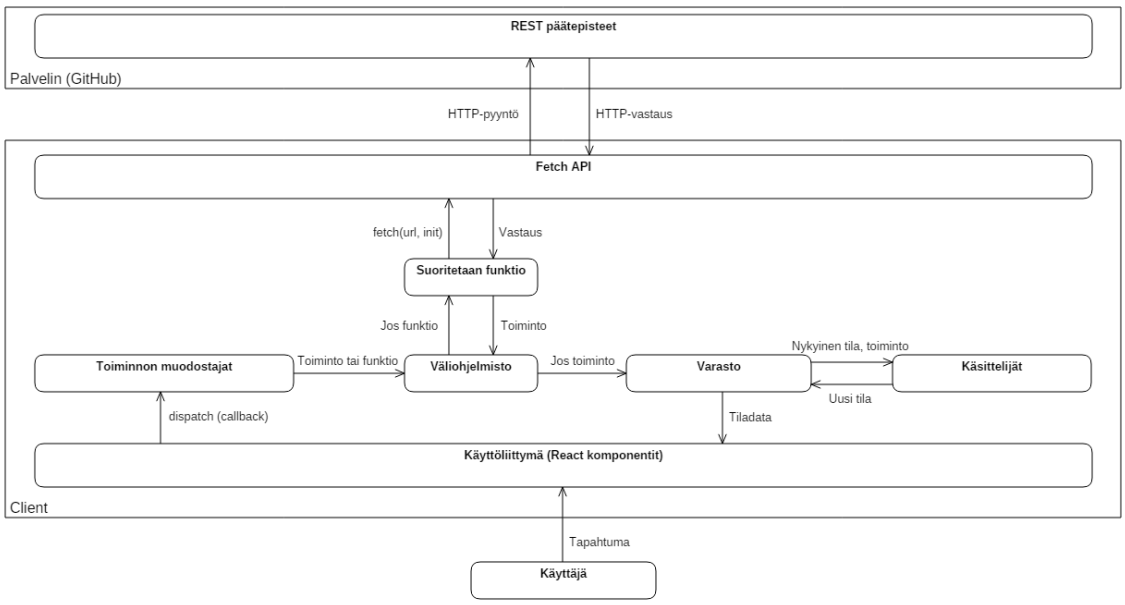
2.2.3 Asynkroninen tiedonkulku

Sovelluksen täytyy pystyä suorittamaan myös asynkronisia operaatioita. Reduxissa asynkronisuuden toteutus perustuu toimintojen lähettämisen viivyttämiseen. Yksinkertaisimmillaan asynkroninen operaatio voidaan suorittaa esimerkiksi timeout-funktion avulla (kuva 7). (StackOverflow 2016.)

```
//Suljetaan ilmoitus, kun 4 sekuntia on kulunut.
setTimeout(() => dispatch(closeNotification()), 4000);
```

Kuva 7. Ilmoituksen sulkeminen asynkronisesti timeout-funktion avulla.

Vaativimmissa asynkronisissa operaatioissa, kuten verkkoresurssien hakemisessa, käytetään apuna Redux Thunk -väliohjelmistoa. Väliohjelmiston avulla asynkroninen logiikka voidaan siirtää toiminnon muodostajille. Kuva 8 havainnollistaa, miten väliohjelmistoa voidaan hyödyntää asynkronisen tiedonkulun toteuttamiseen. Väliohjelmiston avulla toiminnon muodostaja voi lähettää paluuarvonaan funktion, toiminnon sijaan. Toiminnot kulkevat suoraan väliohjelmiston läpi varastolle kuvan 8 mukaisesti, mutta funktiot jäävät väliohjelmiston suoritettavaksi. Toiminnon muodostajan palauttama funktio voi lähettää toimintoja ja suorittaa esimerkiksi API-kutsuja. (Redux 2016i; Redux 2016h.)



Kuva 8. Asynkroninen tiedonkulku sovelluksessa.

Kuva 9 esittää yleisintä asynkronisuuden käyttötarkoitusta toiminnallisessa osuudessa, verkkoresurssien hakemista Gist-palvelusta. Käytännössä kaikki sovelluksen verkkoresurssien hakeminen hoidetaan kuvan 9 esittämällä tavalla. Toiminnon muodostaja lähetetään store.dispatch-metodin välityksellä ja lähetetty toiminnon muodostaja palauttaa funktion, joka suorittaa operaation asynkronisen osuuden. Funktio hakee resurssit Fetch API:n välityksellä, käsittelee vastauksen ja lähettää toimintoja, kun tarvittava data on saatavilla. On tärkeää huomata, että funktion lähettämät toiminnot eivät eroa näkymäkerroksen lähettämistä toiminnoista millään tavalla. Funktiosta lähetetty toiminto noudattaa siis kuvan 6 mukaista synkronista tiedonkulkua (Redux 2016h).

```
function fetchResource() {
    //Toiminnon muodostaja palauttaa funktion, toiminnon sijaan.
    return (dispatch) => {
        //Lähetetään haun alkamisesta ilmoittava toiminto.
        dispatch(requestResource());

        //Lähetetään pyyntö.
        return fetch('https://api.github.com/gists')
            //Käsitellään vastaus.
            .then(response => {
                //Jos pyyntö onnistui, luetaan vastausksen sisältö ja lähetetään se eteenpäin.
                if(response.ok) {
                    response.json().then(json => dispatch(receiveResource(json)));
                //Jos pyyntö epäonnistui, heitetään poikkeus.
                } else {
                    throw new Error(response.status + ' ' + response.statusText);
                }
            }).catch(error => dispatch(fetchFailed(error)));
    };
}
```

Kuva 9. Asynkronisen hakuoperaation suorittava toiminnon muodostaja.

2.3 Työkalut

2.3.1 React

Sovelluksen käyttöliittymän rakennettiin Reactin avulla. React on Facebookin, Instagramin ja aktiivisen kehittäjäyhteisön yhteistyössä kehittämä ja ylläpitämä JavaScript-kirjasto käyttöliittymien rakentamiseen (Facebook 2016a).

Käyttöliittymän rakentaminen Reactilla perustuu komponenttiajatteluun. Komponenttien avulla käyttöliittymän mallintamien voidaan jakaa pieniin itsenäisesti toimiviin, uudelleenkäytettäviin osiin. Komponentteja koostetaan hierarkiaksi, josta käyttöliittymä lopulta muodostuu. (Facebook 2016b.)

Komponentti voidaan ajatella JavaScript-funktiona, joka ottaa vastaan argumentteja (attribuutteja) ja palauttaa kuvaksen käyttöliittymään renderöitävästä sisällöstä. Komponentin palauttama kuvaus muodostuu React-elementeistä. React-elementti on JavaScript-olio, joka kuvaa React-komponentin tai HTML-elementin. React muokkaa DOM:ia komponentin palauttamien React-elementtien pohjalta. (Facebook 2016c.)

Komponenttipohjaisuuden ohella Reactin toimintaan kuuluu keskeisesti virtuaalinen DOM. Virtuaalinen DOM on kevyt abstraktio dokumentin DOM:ista. Kun komponentin data muuttuu, React päivittää käyttöliittymän automaattisesti vain muutoksia sisältävien elementtien osalta vertaamalla komponenttien palauttamia kuvauksia virtuaali-DOM:iin tallennetuihin arvoihin. (Krajka 2016.)

Kuvassa 10 havainnollistetaan, kuinka yksinkertaisella komponenttihierarkialla saadaan kuvattua sovelluksen ylätunniste. Header-komponentti kuvaa lapsikomponenttinsa UserInfon ja NavMenun, jotka puolestaan kuvaavat sovelluksen ylätunnisteen muodostavat HTML-elementit. Kuvantamisen ohella kuvassa 10 esitellään, miten dataa voidaan välittää komponenttien välillä. Komponentit voivat välittää dataa lapsikomponenteilleen argumentteina. Välitetyt argumentit voidaan lukea komponentissa this.props-oliosta. (Facebook 2016c.)

```
class NavMenu extends React.Component {
    render() {
        return (
            <ul className='nav-menu'>
                <li><Link to='/opinnaytetyo'>Listaa gistit</Link></li>
                <li><Link to='/opinnaytetyo/create'>Luo uusi gist</Link></li>
            </ul>
        );
    }
}

class UserInfo extends React.Component {
    render() {
        return (
            <div className='user-info'>
                <img src={this.props.avatarUrl}>
                <p>{this.props.username}</p>
            </div>
        );
    }
}

//Header renderöi lapsikomponenttinsa UserInfo ja NavMenu.
class Header extends React.Component {
    render() {
        return (
            <UserInfo
                username={'TatuPutto'}
                avatarUrl={'https://avatars3.githubusercontent.com/u/5699778?v=3&s=40'}
            />
            <NavMenu />
        );
    }
}

//Renderöidään Header-komponentti DOM-solmuun.
ReactDOM.render(<Header />, document.getElementById('container'));
```

Kuva 10. Esimerkki Reactista.

Attribuuttien ohella komponentit voivat säilyttää sisäistä tilaansa (Facebook 2016c). Komponentin sisäistä tilaa ei tule sekoittaa Reduxin hallinnoimaan sovelluksen tilaan. Sisäinen tiladata on käytössä vain komponentille itselleen ja yleisesti ottaen se kuvaa vain pieniä, paikallisia käyttöliittymän toimintaan liittyviä tiloja, kuten onko alasvetovalikko auki vai kiinni tai mitä tekstikenttään on kirjoitettu. Tarkkaa ohjesääntöä tiladatan jakamisesta Redux-varaston ja komponentin sisäisen tilan välille ei ole. Yleensä päätös täytyykin tehdä tilannekohtaisesti. Selkeyden vuoksi tässä sovelluksessa käyttöliittymän toimintaan liittyvä tiladata pyritään säilyttämään komponenttien sisäisenä tilana ja sovelluksen data Redux-varastossa. (Redux 2016j.)

Tila alustetaan komponentin konstruktori-metodissa asettamalla this.state-muuttujan arvo. Suorituksen aikana komponentti päivittää tilaansa setState-metodilla. Kuvassa 11 havainnollistetaan, miten komponentin sisäistä tilaa

voidaan hyödyntää interaktiivisuuden toteuttamiseen. Komponentin tilaksi alustetaan isOpen-muuttuja, jonka perusteella painikkeen value-attribuutin arvo määritellään ja suodatustoiminnot renderöidään tai jätetään renderöimättä. Kun käyttäjä painaa painiketta, isOpen-muuttujan arvoa muutetaan toden ja epätoden välillä. Komponentin tilan muutos aiheuttaa automaattisesti komponentin uudelleen renderöimisen. Näin DOM:ia päivitetään dynaamisesti vastauksena käyttäjän klikkaukseen. (Facebook 2016c.)

```
class Filters extends React.Component {
    constructor() {
        super();
        this.toggleFilteringOptions = this.toggleFilteringOptions.bind(this);

        //Alustetaan komponentin tila.
        this.state = {isOpen: false};
    }

    //Muutetaan komponentin tilaa.
    toggleFilteringOptions() {
        this.setState({isOpen: !this.state.isOpen});
    }

    render() {
        //Määritellään painikkeen arvo isOpen-muuttujan arvon perusteella.
        const buttonText = this.state.isOpen ? 'Piilota suodattimet' : 'Näytä suodattimet';

        return (
            <div className='filters'>
                <button onClick={this.toggleFilteringOptions}>{buttonText}</button>

                {/*Mallinnetaan suodatustoiminnot, jos isOpen === true*/}
                {this.state.isOpen &&
                    <FilteringOptions />
                }
            </div>
        );
    }
}
```

Kuva 11. Komponentin sisäisen tilan hyödyntäminen.

React-elementtien määrittelyyn käytetään apuna JSX-syntaksia (JavaScript XML). JSX:n avulla React-elementtejä voidaan luoda XML:n tapaisella syntaksilla. JavaScript-lausekkeet erotellaan XML:stä lisäämällä arvon ympärille aaltosulkeet. JSX:n käyttäminen ei ole pakollista, se kuitenkin mahdollistaa React-elementtien määrittelemisen huomattavasti helpommin tulkittavassa

muodossa. Esimerkiksi UserInfo-komponentti ilman JSX-syntaksia näyttäisi kuvan 12 mukaiselta. (Facebook 2016c.)

```
class UserInfo extends React.Component {
    render() {
        return (
            React.createElement('div', {className: 'user-info'},
                React.createElement('img', {src: this.props.avatarUrl}),
                React.createElement('p', {}, this.props.username),
            )
        );
    }
}

//UserInfo-komponentti renderoidaan yliluokassa esimerkiksi näin:
React.createElement(UserInfo, {avatarUrl: 'https://...', username: 'TatuPutto'})
```

Kuva 12. UserInfo-komponentti ilman JSX-syntaksia.

2.3.2 React Router

Sovelluksesta tehtiin SPA-arkkitehtuurin (engl. single-page application) mukainen eli ns. yhden sivun sovellus. Yhden sivun sovelluksessa käyttöliittymä rakentuu yhden HTML-sivun varaan. Kun käyttäjä navigoi sivulla, uuden HTML-sivun lataamisen sijaan tarvittavat resurssit haetaan asynkronisesti ja sivulla esitettävä sisältö korvataan haetulla sisällöllä dynaamisesti. Tätä kautta säästytään kokonaisilta sivun uudelleen lataamisilta, mikä tekee siirtymisestä näkymien välillä huomattavasti nopeampaa ja käyttökokemuksesta saadaan huomattavasti sulavampi. (Wikipedia 2016c.)

SPA-arkkitehtuurin toteuttamisen tueksi valittiin React-pohjaisia sovelluksia varten kehitetty React Router -reitityskirjasto. React Routerilla sovellukselle määritellään reitit (engl. routes) (kuva 13). Jokaiselle reitille määritellään polku sekä React-komponentti. Kun käyttäjä syöttää reittiä vastaavan URL-osoitteen, reitille määritelty komponentti renderöidään. Polun ja komponentin ohella reitille voidaan määritellä funktio, joka suoritetaan reittisiirtymän yhteydessä. (Dabit 2016; React Router 2016.)

```
<Route path='opinnaytetyo' component={Root}>
    <IndexRoute component={ListingPage} onEnter={fetchGistsOnEnter} />
    <Route path='gists' component={ListingPage} onEnter={fetchGistsOnEnter} />
    <Route path='starred' component={ListingPage} onEnter={fetchGistsOnEnter} />
    <Route path='discover(/:page)' component={ListingPage} onEnter={fetchGistsOnEnter} />
    <Route path='search(/:user)' component={ListingPage} onEnter={fetchGistsOnEnter} />
    <Route path='gist/:gistId' component={SingleGist} onEnter={fetchSelectedGistOnEnter} />
    <Route path='edit/:gistId' component={EditGist} onEnter={fetchSelectedGistOnEnter} />
    <Route path='create' component={CreateGist} />
</Route>
```

Kuva 13. Sovelluksen reittien määrittely.

SPA-arkkitehtuuri tuo mukanaan kourallisen haasteita. Tämän projektin kannalta keskeisiä ongelmia ovat varsinkin sivuhistorian hallinta ja resurssien linkitettävyys. (Wikipedia 2016c.) Edellä mainitut ongelmat ratkaistaan HTML5 History API:n avulla. History API:n avulla selaimen historiaa ja URL-osoitetta voidaan manipuloida ohjelmallisesti pushState- sekä replaceState-metodien välityksellä. React Router hyödyntää edellä mainittuja metodeja automaattisesti. Kun React Routerin API:a käytetään navigointiin, selaimen osoitekentässä näytettävä URL-osoite saadaan pidettyä ajan tasalla sovelluksessa kulloinkin näytettävän sisällön kanssa. Siirtymien yhteydessä myös selaimen historiaa päivitetään, joten historiaa voidaan hyödyntää samalla tavalla kuin perinteisessä verkkosovelluksessa. (React Router 2016.)

2.3.3 ACE

Koodileikkeiden esittämiseen ja käsittelemiseen päätettiin käyttää suoraan sivulle upotettavaa koodieditoria. Tärkeimmät kriteerit koodieditorin valitsemiselle olivat syntaksin korostus, käytettävyys ja ulkoasun kustomointi. Kriteerien pohjalta koodieditoriksi valittiin Ajax.org Cloud9 editor (ACE). ACE on JavaScript-pohjainen koodieditori, joka vastaa toiminnallisuuksiltaan, käytettävyydeltään ja suorituskyvyltään natiiveja koodieditoreja. (Ace 2016; Wikipedia 2016d.)

ACE tarjoaa laajan valikoiman asetuksia, joita hyödyntämällä editorista saadaan joustava kokonaisuus. Luotavaan editoriin voidaan esimerkiksi asettaa alustava arvo, onko editori vain lukutilassa, montako riviä näytetään jne. Näiden

asetuksien avulla ACE-ilmentymän luominen pystyttiin sisällyttämään React-komponenttiin, jota voitiin uudelleenkäyttää sovelluksen näkymien välillä välittämällä komponentille erilaiset argumentit (kuva 14).

```
class Editor extends React.Component {
    //Muodostetaan editori <div>-elementtiin,
    //mikäli komponentti liitettiin onnistuneesti DOM:iin.
    componentDidMount() {
        const {editorId, value, language, isReadOnly} = this.props;

        const editor = ace.edit(editorId);
        editor.$blockScrolling = Infinity;
        editor.setTheme('ace/theme/eclipse');
        editor.getSession().setMode('ace/mode/' + language);
        editor.getSession().setUseWorker(false);
        editor.setShowPrintMargin(false);
        editor.setReadOnly(isReadOnly);
        editor.setOptions({fontSize: '14px'});

        if(value) {
            const lines = value.split('\n').length + 1;
            const valueWithNewLine = value.endsWith('\n') ? value : value + '\n';
            editor.setValue(valueWithNewLine);

            if(isReadOnly) {
                editor.setOptions({maxLines: lines});
            } else {
                editor.setOptions({minLines: 10});
                editor.setOptions({maxLines: 60});
            }
            editor.selection.moveTo(0);
            ...
        }
    }

    render() {
        return <div id={this.props.editorId}></div>;
    }
}
```

Kuva 14. ACE-ilmentymän näkymään luova Editor-komponentti.

2.3.4 Webpack

Reduxin ja Reactin käyttöönottamisen yhteydessä riippuvuuksien määrä alkoi kasvaa hyvin nopeasti, uusien kirjastojen sekä tehokkaamman lähdekoodin modularisoinnin seurauksena. Pian riippuvuuksien hallitseminen manuaalisesti osoittautui liian työlääksi. Tässä vaiheessa Webpack otettiin mukaan kehitysympäristöön.

Webpack on moduuliniputtaja (engl. module bundler), jonka avulla riippuvuuksia sisältävät moduulit voidaan paketoida yhdeksi tiedostoksi. Käytännössä Webpack mahdollistaa front-endin muodostavan sisällön koostamisen yhdeksi JavaScript-tiedostoksi. Sen sijaan että jokainen riippuvuus liitettäisiin manuaalisesti HTML-sivulle, sivulle ladataan vain yksi minimoitu JavaScript-tiedosto, joka sisältää kaiken tarvittavan sisällön. (Webpack 2016a.)

Sovelluksen JavaScript-sisältö on kirjoitettu ES6-syntaksia käyttäen. ES6-tuki selaimissa on kuitenkin vielä melko heikkoa. Tukiongelmat ratkaistiin lisäämällä Webpackin toiminnallisuuksia laajentavia lataajia (engl. loaders). Lataajien avulla Webpack voi esikäsitellä tiedostoja ja esimerkiksi muuttaa tiedoston lähdekoodin ohjelmointikielestä toiseen. (Webpack 2016b.) Jotta ES6-syntaksia hyödyntävistä tiedostoista saatiin yhteensopivia mahdollisimman monen selainversion kanssa, tiedostojen ES6-syntaksia käyttävät osiot käännettiin ES5-muotoon Babel-lataajan avulla. Babel-lataajaa käytettiin myös React-komponenteissa käytettävän JSX:n JavaScriptiksi kääntämiseen. Babel-lataajan ohella sovelluksessa käytettiin myös CSS- ja JSON-lataajia tyylitiedostojen ja JSON-resurssien käyttämiseksi paketin sisältämissä moduuleissa.

2.3.5 Webpack dev server

Sovelluksen kehitysvaiheessa palvelinympäristönä käytettiin Webpack dev serveriä. Webpack dev server on pieni Node.js Express -palvelin, jonka avulla Webpackilla koostettua pakettia voidaan päivittää vain muutoksia sisältävien moduulien osalta. Webpack dev server tarkkailee paketin sisältämiä moduuleja ja päivittää pakettia automaattisesti vain muutoksia sisältävien moduulien osalta, kun tarkkailtavissa moduuleissa tapahtuu muutoksia. Näin vältytään koko paketin uudelleen koostamiselta aina kun moduuleihin tehdään muutoksia. Osittaisen paketin koostamisen lisäksi hyödynnettiin selaimen automaattista päivitystä. Kun pakettia päivitetään, selain päivitetään automaattisesti vastaamaan pakettiin tehtyjä muutoksia. (Webpack 2016c; Webpack 2016d.)

2.4 Reduxin integroiminen Reactin kanssa

Redux yhdistetään Reactiin välittämällä React-komponentille ilmentymä varastosta. Ilmentymän kautta varaston API on komponentin käytettävissä. Ajatuksena on rekisteröidä komponentti kuuntelemaan tilan muutoksia subscribe-metodin avulla. Tilan muutoksen yhteydessä rekisteröityä komponenttia kutsutaan, jolloin komponentti voi lukea päivitetyn tilan getState-metodilla. Edellä mainittua logiikkaa ei kirjoiteta itse, vaan komponentista, joka halutaan yhdistää Reduxiin, luodaan uusi ilmentymä connect-funktiolla kuvan 15 mukaisesti. Luotu ilmentymä sisältää komponentin päivityslogiikan tehokkaasti toteutettuna. (Redux 2016k.)

```
connect(mapStateToProps, mapDispatchToProps)(GistList);
```

Kuva 15. GistList-komponentin yhdistäminen Reduxiin.

Connect-funktiolla on yksi pakollinen argumentti, mapStateToProps-funktio. MapStateToProps määrittää mitä sovelluksen tilasta luetaan ja miten luettu data muutetaan komponentin attribuuteiksi (Kuva 16). (Redux 2016k.)

```
//Luetaan listausnäkymän tarvitsema tiladata
//ja määritellään miten data muutetaan attribuuttidataksi.
function mapStateToProps(state) {
    return {
        gists: {
            ...state.gists,
            items: filterByLanguage(state.filters.languages, state.gists.items)
        },
        activeGist: state.activeGist,
        filters: state.filters,
        pagination: state.pagination,
        userId: state.user.id
    };
}
```

Kuva 16. Tiladatan muuttaminen ListingPage-komponentin attribuuteiksi.

Toimintojen lähettämiseksi connect-funktiolle määritellään toinen argumentti, mapDispatchToProps-funktio (kuva 17). MapDispatchToProps palauttaa attribuutteja, jotka sisältävät callback-funktiot halutun toiminnon lähettämiseen.

Toiminto voidaan lähettää esimerkiksi suorittamalla callback-funktion sisältävä ominaisuus vastauksena onClick-tapahtuman (Kuva 18). (Redux 2016k.)

```
//Määritellään listausnäkymän toiminnot.
function mapDispatchToProps(dispatch) {
    return {
        //Suodatustoiminnot.
        filteringActions: {
            addFilter: (language) => dispatch(addFilter(language)),
            removeFilter: (language) => dispatch(removeFilter(language)),
            refresh: () => dispatch(refresh(fetchParams.method, fetchParams.page))
        },

        //Aktiivisen gistin toiminnot.
        gistActions: {
            setActive: (id) => {
                if(id !== activeId) {
                    dispatch(fetchSelectedGist(id));
                }
            },
            forkGist: (id) => dispatch(checkIfForked(id)),
            toggleStarredStatus: (isStarred, id) => {
                if(isStarred) {
                    dispatch(unstarGist(id));
                }
                else {
                    dispatch(starGist(id));
                }
            },
            deleteGist: (id) => {
                if(confirm('Haluatko varmasti poistaa tämän gistin?')) {
                    dispatch(deleteGist(id));
                }
            }
        }
    };
}
```

Kuva 17. Callback-funktioiden sisällyttäminen ListingPage-komponentin attribuutteihin.

```
<li className={isActive} id={this.props.id} onClick={() => this.props.setActive(this.props.id)}>
```

Kuva 18. Toiminnon lähettäminen lapsikomponentista callback-funktion välityksellä.

Reduxiin yhdistetyt komponentit pyritään pitämään mahdollisimman vähäisinä. Tämän sovelluksen tapauksessa Reduxiin yhdistettiin näkymien pääkomponentit. Jaottelu ei ole optimaalinen, mutta se tekee sovelluksen logiikasta helpommin ymmärrettävää ja kuvattavaa, kun kaikki näkymään tuotava data ja toiminnallisuudet löytyvän yhdestä paikasta.

3 Sovelluksen suunnittelu ja toteutus

3.1 Kehitysprosessi

Sovelluksen kehittäminen aloitettiin yksinkertaisella Java-ympäristöllä. Ensimmäinen versio oli Eclipse-kehitysympäristön konsolin kautta käytettävä sovellus, joka haki ja lisäsi sisältöä Gist-palveluun API:n välityksellä. Kun perustoiminnot oli luotu, alettiin kehittää graafista käyttöliittymää. Ensimmäiset versiot tehtiin JSP-tekniikalla (JavaServer Pages). Hyvin nopeasti kuitenkin huomattiin, että SPA-arkkitehtuurin mukaisen sovelluksen toteuttaminen JSP-pohjaisella käyttöliittymällä olisi vaikeaa. Tässä vaiheessa SPA-arkkitehtuurin toteutusta alettiin tutkia tarkemmin ja vertailemaan tarvittavia työkaluja. Vaihtoehtoina pohdittiin mm. Angularia, Angular 2:ta sekä Reactia, joista lopulta päädyttiin valitsemaan React.

Reactin ja React Routerin avulla SPA-arkkitehtuurin mukainen sovellus saatiin rakennettua nopeasti. Kommunikointi Gist-palvelun kanssa hoidettiin tässä vaiheessa vielä palvelinsovelluksen kautta ja data haettiin käyttöliittymään suoraan React-komponenteista, palvelinsovelluksen paljastaman API:n välityksellä. En kuitenkaan ollut tyytyväinen tähän toteutukseen, koska palvelimen kautta resurssien kuljettaminen lisäsi selainsovelluksen ja Gist-palvelun väliin käytännössä täysin turhan vaiheen. Niinpä Gist-palvelun kanssa kommunikointi päätettiin siirtää selainsovellukseen. Tarvittava logiikan toteuttamisen tueksi selainsovelluksella otettiin Redux. Suurin osa bisneslogiikasta siirrettiin pois näkymäkerroksesta ja toteutettiin Redux-arkkitehtuurin mukaisesti omissa moduuleissaan.

3.2 Käyttöliittymän suunnittelu

Käyttöliittymää ei projektin alussa lähdetty suunnittelemaan kovin tarkalla tasolla. Suunnitelmavaiheessa laadittiin karkea hahmotelma sivuston asettelusta. Asettelun ohella suunniteltiin myös, millaisia näkymiä sovellukselle määriteltyjen

toiminnallisuuksien toteuttamiseksi tarvitaan ja miten toiminnot jakautuvat näkymien välille.

Asettelussa pyrittiin ottamaan huomioon etenkin koodileikkeiden mahdollisimman hyvä luettavuus. Käytännössä luettavuuden varmistaminen tarkoitti sitä, että koodileikkeet esittäville editoreille varattiin jokaisessa näkymässä tarpeeksi tilaa. Tätä ajatellen päädyttiin asetteluun, jossa sivun rakenne koostuu ylätunnisteesta, sisältöelementistä sekä alatunnisteesta. Näin sisältöelementille jäi käytettäväksi koko ikkunan leveys ja varmistuttiin siitä, että koodileikkeet ovat helposti luettavissa pienemmilläkin näyttökooilla.

Näkymiä alettiin suunnitella sovelluksen toiminnallisuuksien pohjalta. Näkymät jaettiin sen pohjalta, mitä toiminnallisuuksia mikäkin näkymä sisältäisi. Käyttöliittymä päädyttiin jakamaan neljään erilliseen näkymään: listausnäkymään, luontinäkymään, muokkausnäkymään sekä yksittäisen gistin näkymään. Näkymien suunnitteluun haettiin ideoita useista eri verkkosivuista. Vaikutteita otettiin mm. GitHubin sekä YouTuben käyttöliittymistä, joita suunnitteluvaiheessa yhdisteltiin omiin ideoihini. Tätä kautta suunnitteluun käytettävää työpanosta saatiin kevennettyä. Ideoiden pohjalta näkymistä tehtiin karkeat hahmotelmat.

Näkymien suunnittelussa kiinnitettiin erityisesti huomiota käytettävyyteen. Tavoitteena oli saada suunniteltua näkymät siten, että koodileikkeet ovat aina helposti luettavissa, editorit asemoituvat mielekkäästi suhteessa muuhun sisältöön ja kullekin näkymälle määritellyt toiminnot esitetään selkeästi ja niitä on helppo käyttää. Käytettävyyden ohella näkymien väliltä pyrittiin löytämään rakenteellisia yhtäläisyyksiä, jotta komponentteja voitiin uudelleen käyttää mahdollisimman tehokkaasti.

3.3 Kommunikointi Gist API:n kanssa

Selaimesta pyyntöjen lähettämiseen käytettiin Fetch API:a. Fetch valittiin perinteisen XMLHttpRequestin sijaan, Promise-pohjaisen API:nsa takia.

Promisejen avulla verkkoresurssien hakeminen pystyttiin sisällyttämään helposti Reduxin asynkroniseen tiedonkulkuun. (Mozilla 2016; Redux 2016h.)

Sen lisäksi, että Promise-pohjainen API on yhteensopiva Reduxin kanssa, sitä voitiin hyödyntää tehokkaasti hakuprosesseille yhteisen logiikan uudelleenkäyttämiseen. Työskentely ulkoisen API:n kanssa edellytti samojen vaiheiden toistamista lähestulkoon jokaisessa hakuoperaatiossa. Nopeasti huomattiin, että pyynnön muodostaminen ja lähettäminen sekä vastauksen käsittelylogiikka säilyivät lähes muuttumattomana operaatioiden välillä. Niinpä edellä mainittu logiikka päätettiin siirtää omiin funktioihinsa (kuva 19).

```
//Tarkistetaan onnistuiko pyyntö.
export function checkStatus(response) {
    if(response.ok) {
        return Promise.resolve(response);
    } else {
        throw new Error(response.status + ' ' + response.statusText);
    }
}

//Luetaan vastauksen sisältö.
export function readJson(response) {
    return response.json();
}

export function sendRequest(url, httpMethod, content = null) {
    let fetchInit;

    //Määritellään pyynnön otsikot ja sisältö (POST, PATCH, PUT).
    if(content) {
        fetchInit = {
            method: httpMethod,
            body: content,
            headers: {
                'Accept': 'application/json',
                'Content-Type': 'application/json',
                'Content-length': content.length,
                'Authorization': 'token ' + accessToken,
            },
        };
    //Määritellään pyynnön otsikot, kun pyynnöllä ei ole sisältöä (GET, DELETE).
    } else {
        method: httpMethod,
        headers: {
            'Accept': 'application/json',
            'Authorization': 'token ' + accessToken,
        },
    }

    //Lähetetään pyyntö.
    return fetch(url, fetchInit);
}
```

Kuva 19. Hakuprosesseille yhteisen logiikan suorittaminen erillisissä funktioissa.

Pyynnön muodostamisen ja lähettämisen sekä vastauksen käsittelyn hoitavat funktiot määriteltiin muotoon, jossa niitä pystyttiin hyödyntämään osana Promise-ketjua. Näin funktiot voitiin sisällyttää saumattomasti hakuprosesseihin ketjuttamalla Promiseja Promise.then-metodin avulla (kuva 20). Hakuprosesseissa hyödynnettävä Promise-ketju jakautuu viiteen vaiheeseen:

1. SendRequest-funktio lähettää pyynnön ja ottaa vastauksen onnistuneesti vastaan.
2. CheckStatus-funktio tarkistaa vastauksen tilan. Jos status-koodi on väliltä 200 - 299, pyyntö tulkitaan onnistuneeksi.

3. ReadJson-funktio lukee vastauksen sisällön.
4. Määritellään mitä vastauksen sisällöllä tehdään.
5. Jos vaiheissa 1, 2 tai 3 tapahtuu virhe, siirrytään suoraan catch-lauseeseen, jossa virhetilanne käsitellään.

```
function fetchResource() {
    return (dispatch) => {
        dispatch(requestResource());
        return sendRequest('https://api.github.com/gists', GET)
            .then(checkStatus)
            .then(readJson)
            //Jos pyyntö onnistui ja vastauksen sisältö luettiin onnistuneesti.
            .then((data) => dispatch(receiveResource(data)))
            //Jos pyyntö epäonnistui tai vastauksen sisältöä ei onnistuttu lukemaan.
            .catch((error) => dispatch(fetchFailed(error)));
    };
}
```

Kuva 20. Uudelleenkäytettävän käsittelylogiikan hyödyntäminen hakuoperaatiossa Promiseja ketjuttamalla.

Kun pyynnön ja vastauksen käsittelylogiikka siirrettiin erillisiin funktioihin, säästyttiin turhalta koodin toistamiselta ja hakuprosessit suorittavat funktiot saatiin pidettyä huomattavasti selkeämpinä. Lisäksi saavutettiin rakenne, jossa hakuoperaatioita suorittavissa funktioissa voidaan keskittyä määrittelemään vain mitä vastauksen sisällöllä tehdään ja miten virhetilanteet käsitellään.

3.4 Sovelluksen alustus

käyttäjän syöttämä osoite saapuu ensin palvelinsovellukselle. Palvelinsovelluksesta pyyntö ohjataan eteenpäin index.html-sivulle, mikäli URL-osoitteen polkunimi vastaa jotakin seuraavista: /, gists, starred, discover, search, gist, edit tai create. Kun käyttäjä saapuu index.html-sivulle, JavaScript-paketti ladataan ja selainsovellus käynnistyy.

Selainsovelluksen käynnistyksen yhteydessä suoritetaan muutama tärkeä toimenpide. Ensimmäiseksi tarkistetaan löytyykö tarvittavat käyttäjätiedot evästeistä. Käyttäjä ohjataan kirjautumisnäkymään, mikäli tietoja ei löydy tai käyttöavain ei ole enää voimassa. Jos asianmukaiset käyttäjätiedot löytyvät

evästeistä Redux varastosta luodaan ilmentymä. Varasto-ilmentymän luomisen jälkeen suoritetaan react-dom-kirjaston render-metodi, joka määrittää mitä DOM:iin viime kädessä renderöidään.

Selainsovelluksen reitit määritellään ReactDOM.render-metodin sisällä, jotta se voi renderöidä URL-osoitetta vastaavan näkymän (kuva 21). Reitityksessä käytetään sisäkkäisiä reittejä (engl. nested routes). Sovelluksen ainoa ylemmän tason reitti on sidottu sovelluksen URL-osoitteen juureen (/opinnaytetyo), tälle reitille määritelty Root-komponentti renderöidään siis käytännössä aina. Ylemmän tason reitin löytämisen jälkeen etsitään URL-osoitteen polkunimeä vastaava lapsireitti (engl. child route), jolle määritelty komponentti asetetaan Root-komponentin lapsikomponentiksi. Esimerkiksi käyttäjän syöttäessä osoitteen http://localhost:8080/opinnaytetyo/create, valitaan Root-komponentti ja lapsikomponentiksi create-reitille määritelty CreateGist-komponentin, joka muodostaa näkymän gistin luontia varten. Mikäli URL-osoite kohdistuu sovelluksen juureen, lapsikomponentiksi valitaan IndexRoute-reitille määritelty komponentti.

```
ReactDOM.render(
    <Provider store={store}>
        <Router history={browserHistory}>
            <Route path='opinnaytetyo' component={Root}>
                <IndexRoute component={ListingPage} onEnter={fetchGistsOnEnter} />
                <Route path='gists' component={ListingPage} onEnter={fetchGistsOnEnter} />
                <Route path='starred' component={ListingPage} onEnter={fetchGistsOnEnter} />
                <Route path='discover(/:page)' component={ListingPage} onEnter={fetchGistsOnEnter} />
                <Route path='search(/:user)' component={ListingPage} onEnter={fetchGistsOnEnter} />
                <Route path='gist/:gistId' component={SingleGist} onEnter={fetchSelectedGistOnEnter} />
                <Route path='edit/:gistId' component={EditGist} onEnter={fetchSelectedGistOnEnter} />
                <Route path='create' component={CreateGist} />
            </Route>
        </Router>
    </Provider>,
    document.getElementById('container')
);
```

Kuva 21. ReactDOM.render-metodi.

Kun osoitetta vastaavat komponentit on löydetty, muodostetaan näkymä. Jokaiseen näkymään sisällytetään ylätunniste, ei-intrusiivinen ponnahdusikkuna ilmoitusten välittämiseksi käyttäjälle sekä lapsikomponentista muodostuva runko. Suunnitelmasta poiketen alatunnistetta ei vielä tähän versioon sisällytetty. Haluttu rakenne saatiin muodostettua kovakoodaamalla Root-komponentin

render-metodi renderöimään Header- ja Notifications-komponentit. Jotta näkymän sisältöä voidaan päivittää dynaamisesti, lapsikomponentti tuodaan Root-komponentille children-argumentin välityksellä (kuva 22).

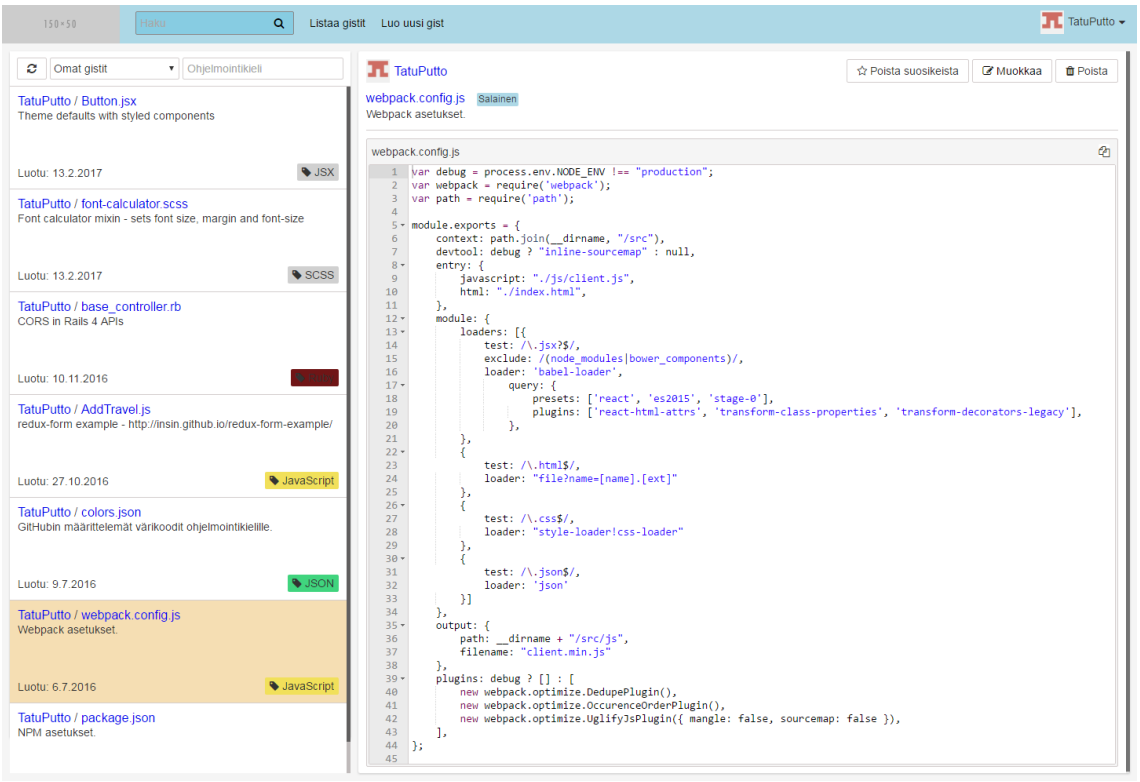
```
class Root extends React.Component {
    render() {
        return (
            <div className='content'>
                <Header />
                <Notifications />
                {this.props.children}
            </div>
        );
    }
}
```

Kuva 22. Näkymän muodostaminen Root-komponentissa.

Reittien määrittelyn lisäksi varastosta luotu ilmentymä levitetään komponenteille ReactDOM.render-metodissa. Komponentit tarvitsevat ilmentymän varastosta, jotta ne voivat kommunikoida varaston kanssa sovelluksen suorituksen aikana. Ilmentymä levitetään komponenteille käärimällä ne kuvan 21 mukaisesti react-redux-kirjaston Provider-komponentilla. Provider saa argumenttinaan varaston ilmentymän, jonka se välittää automaattisesti käärimilleen komponenteille.

3.5 Gistien listausnäkymä

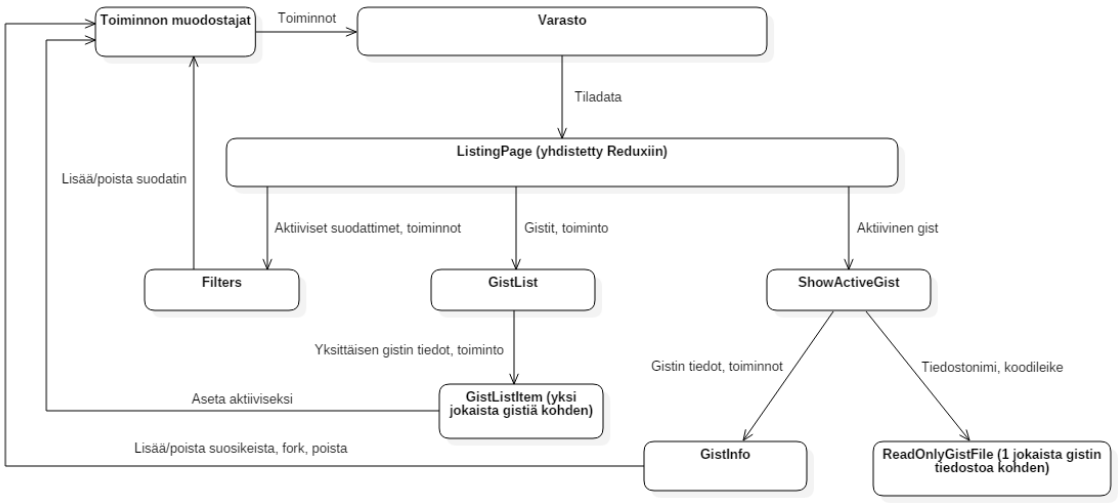
Sovelluksen etusivuna toimii gistien listausnäkymä (kuva 23). Listausnäkymää käytetään käyttäjän gistien listaamiseen sekä discover-toiminnallisuuteen. Listausnäkymän tarkoituksena on esittää hakuehtoja vastaavat gistit helposti selattavassa ja luettavassa muodossa. Käyttäjän tulee myös päästä vaivattomasti käsiksi gistien hallintatoimintoihin.



Kuva 23. Gistien listausnäkymä.

3.5.1 Rakenne

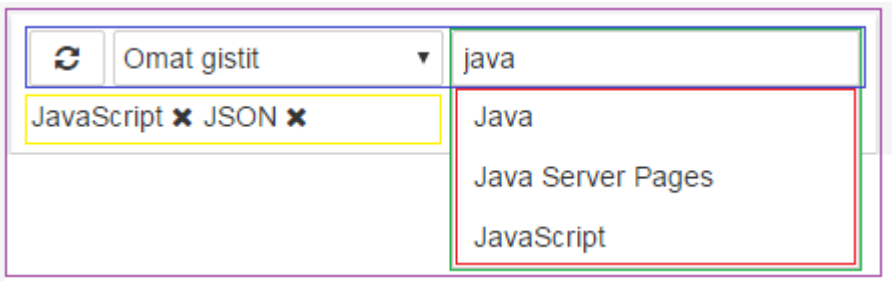
Näkymän pohjana toimii ListingPage-komponentti, joka muodostaa näkymän lapsikomponenteistaan (kuva 24). Tiedonkulku haluttiin säilyttää mahdollisimman yksinkertaisena, joten ListingPage toimii myös näkymän ainoana Reduxiin yhdistettynä komponenttina. ListingPage lukee tarvittavan tiladatan varastosta aina sovelluksen tilan päivittyessä ja välittää luetun datan argumentteina lapsikomponenteilleen. Tiladatan välittämisen lisäksi ListingPage muodostaa varastolle lähetettävät toiminnot ja välittää niiden kutsumiseen tarvittavat callback-funktiot lapsikomponenteilleen.



Kuva 24. Listausnäkymän rakenne ja tiedonkulku yksinkertaistettuna.

Näkymän vasen palsta muodostetaan Filters- ja GistList-komponenteista. Filtrers-komponentti muodostaa listan yläpuolelle elementit listan päivittämiseen, hakutyypin vaihtamiseen sekä gistien suodattamiseen ohjelmointikielen perusteella. Listan hallintatoiminnot muodostuvat komponenteista kuvan 25 mukaisesti:

- Filters (violetti): koostaa kokonaisuuden lapsikomponenteistaan.
- FilteringOptions (sininen): koostaa hallintatoiminnot muodostavat komponentit.
- FilterByLanguage (vihreä): muodostaa suodatustoiminnon tekstikentästä ja Suggestions-komponentista.
- Suggestions (punainen): muodostaa listan ehdotuksista.
- ActiveFilters (keltainen): esittää kulloinkin aktiivisena olevat suodattimet.

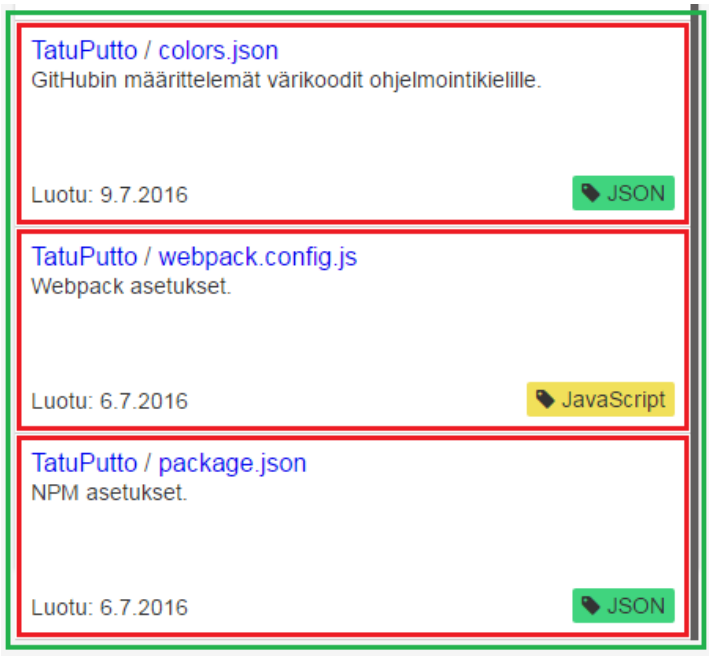


Kuva 25. Listan hallintatoimintojen komponenttihierarkia.

Käyttäjä voi suodattaa listassa näytettäviä gistejä ohjelmointikielen perusteella. Käyttäjä syöttää ohjelmointikielen tekstikenttään, jota vastaavat gistit halutaan

nähdä listassa. Käyttäjälle näytetään ehdotuksia syötettyä merkkijonoa vastaavista ohjelmointikielistä. Käyttäjä lisää suodattimen painamalla enter-näppäintä tai klikkaamalla jotain annetuista ehdotuksista. Aktiiviset suodattimet esitetään p-elementtinä suodatustoimintojen alapuolella. Käyttäjä voi poistaa suodattimen painamalla luotua p-elementtiä.

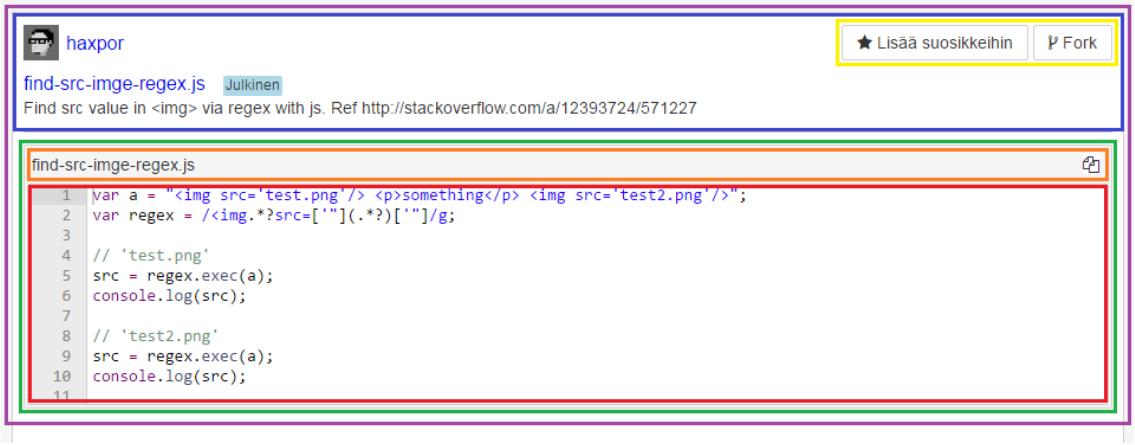
GistList kuvaa ul-elementin, johon listattavien gistien tiedot sijoitetaan li-elementtien muodossa (kuva 26). GistList käy gistin tiedot sisältävän taulukon läpi ja luo jokaista taulukon indeksiä kohden yhden ilmentymän lapsikomponentistaan GistListItem. GistListItem kuvaa li-elementin välitettyjen argumenttien pohjalta.



Kuva 26. Listausnäkymän vasemman palstan komponenttihierarkia.

Näkymän oikeaan palstaan kuvataan aktiiviseksi asetetun gistin tiedot, tiedostot sekä painikkeet toimintojen laukaisemiseksi. Käyttäjälle näytettävät toiminnot määritetään sen mukaan, onko käyttäjä gistin omistaja vai ei. Jos käyttäjä ei ole gistin omistaja, näytetään toiminnot gistin suosikiksi asettamiseen ja kopioimiseen. Jos käyttäjä on gistin omistaja, näytetään toiminnot gistin suosikiksi asettamiseen, muokkaamiseen ja poistamiseen. Kuva 27 esittää miten näkymän kuvaaminen jaettiin komponenttien välille:

- ShowActiveGist (violetti): koostaa näkymän oikean puolen lapsikomponenteistaan.
- GistInfo (sininen): kuvaa gistin tiedot sekä painikkeet kuvaavan lapsikomponentin.
- GistActions tai GistActionsOwner (keltainen): painikkeet toimintojen laukaisemiseksi.
- ReadOnlyGistFile (vihreä): koostaa kirjoitussuojatun tiedostokentän FileInfoWithLink- ja Editor-komponenteista.
- FileInfoWithLink (oranssi): kuvaa tiedostonimen ja painikkeen, jonka avulla koodileike voidaan kopioida leikepöydälle.
- Editor (punainen): Muodostaa kirjoitussuojatun ACE-instanssin.



Kuva 27. Listausnäkymän oikean palstan komponenttihierarkia.

3.5.2 Gistien hakeminen

Listattavien gistin hakeminen aloitetaan automaattisesti käyttäjän saapuessa näkymään. Käyttäjän syöttämän URL-osoitteen perusteella määritellään, minkä tyyppisiä gistejä haetaan. Näkymään voidaan hakea käyttäjän gistit, käyttäjän suosikkeihin asetetut gistit, uusimmat muiden käyttäjien julkiset gistit (discover-toiminnallisuus) tai tietyn käyttäjän julkiset gistit (kuva 28). Käyttäjä voi vaihtaa hakutyyppiä muuttamalla pudotusvalikon arvoa (Omat gistit, Suosikit tai Discover), syöttämällä käyttäjänimen hakukenttään tai syöttämällä hakutyyppiä vastaavan URL-osoitteen. Käyttäjä voi myös halutessaan päivittää listan painamalla päivitys-painiketta.

```
//Haetaan hakuehtoja vastaavat gistit näkymään saavuttaessa.
export function fetchGistsOnEnter(nextState) {
    //Määritetään polkunimen perusteella mitä haetaan.
    let fetchMethod = nextState.location.pathname.match(/gists|starred|discover|search/g);
    fetchMethod = fetchMethod == null ? 'gists' : fetchMethod[0];

    //Haetaanko gistit, vai käytetäänkö välimuistista löytyviä gistejä.
    if(shouldFetch(store.getState(), fetchMethod, nextState.params.page)) {
        if(fetchMethod === 'discover') {
            return store.dispatch(fetchGists('discover', nextState.params.page));
        } else if(fetchMethod === 'search') {
            return store.dispatch(fetchGists('search', null, nextState.params.user));
        }
        return store.dispatch(fetchGists(fetchMethod));
    }
}
```

Kuva 28. Haun aloittaminen näkymään saavuttaessa.

Ennen haun aloittamista tarkistetaan, täytyykö uusia gistejä ladata, vai voidaanko käyttää välimuistiin tallennettuja gistejä. Mikäli välimuistista ei löydy gistejä tai löytyneet gistit eivät ole enää ajan tasalla (hausta kulunut yli minuutti) tai hakutyyppi on eri, gistien hakeminen aloitetaan. Gistit haetaan lähettämällä GET-tyyppinen pyyntö määriteltyyn päätepisteeseen (kuva 29). Hakutyypistä riippuen päätepisteenä voi olla /gists, /gists/starred, /gists/public tai /users/:user/gists.

```
export function fetchGists(fetchMethod, pageNumber = 1, user = null) {
    return (dispatch) => {
        //Mitätöidään aktiivinen gist.
        dispatch(invalidateGist());
        //Ilmoitetaan haun alkamisesta.
        dispatch(requestGists(fetchMethod));

        //Lähetetetään pyyntö ja jäädään odottamaan vastausta.
        //Päätepisteenä voi olla: /gists, /gists/starred, /gists/public tai /users/:user/gists.
        return sendRequest(determineEndpoint(fetchMethod, pageNumber, user), 'GET')
            .then(checkStatus)
            .then(readJson)
            .then((data) => {
                //Parsitaan vastauksen sisältö ja lähetetään parsittu data varastolle.
                const parsedGists = parseMultipleGistsJson(data);
                dispatch(receiveGists(parsedGists));
                //Haetaan ensimmäisen gistin tarkemmat tiedot.
                dispatch(fetchSelectedGist(parsedGists[0].id));

                //Päivitetään sivutus jos hakutyyppinä on discover.
                if(fetchMethod === 'discover') {
                    dispatch(updatePagination(pageNumber));
                }
            }).catch((error) => dispatch(gistsFetchFailed(error.message)));
        };
}
```

Kuva 29. Listattavien gistien hakeminen.

Gistien hakeminen hoidetaan asynkronisesti. Jotta käyttäjä voi havaita haun olevan käynnissä, täytyy käyttöliittymää päivittää hakuprosessin aikana. Käyttöliittymälle viestitään hakuprosessin tilanteesta isFetching-muuttujan välityksellä (kuva 30). Mikäli isFetching-muuttujan arvo on tosi ja listassa ei ole sisältöä, näkymään renderöidään latausindikaattori. Jos listassa on sisältöä edellisen haun jäljiltä, listan opasiteettia lasketaan. Epätosi arvo viestii hakuprosessin päättymisestä, jolloin listassa näytetään haun tuloksena löydetyt gistit tai virheilmoitus.

```
{fetchError &&
    <p>Gistien hakemisessa tapahtui virhe ({fetchError}).</p>
}
{isFetching && listItems.length === 0 && !fetchError &&
    <div className='loading'></div>
}
{!isFetching && listItems.length === 0 && !fetchError &&
    <p>Hakuehtoja vastaavia gistejä ei löytynyt.</p>
}
{listItems.length > 0 && !fetchError &&
    <div style={{opacity: isFetching ? 0.5 : 1}}>
        <ul>
            {listItems}
        </ul>
    </div>
}
```

Kuva 30. GistList-komponentin sisällön määrittäminen.

Haussa on kolme vaihetta, joihin saavuttaessa lähetetään toimintoja. Haun alkaessa lähetetään requestGists- sekä invalidateGist-toiminnot. RequestGists ilmoittaa haun alkamisesta ja minkä tyyppisiä gistejä ollaan hakemassa (kuva 31). InvalidateGist ilmoittaa, että nykyinen aktiivinen gist tulisi mitätöidä ja tätä kautta näkymän oikea puoli tyhjentää.

```
//@actions.js
//Ilmoitetaan haun alkamisesta.
function requestGists(fetchMethod) {
    return {type: 'FETCH_GISTS_REQUEST', fetchMethod};
}

//@gists.js - gistien osalta tilaa hallitseva käsittelijä-funktio.
//Päivitetään sovelluksen tila vastaamaan haun tilannetta.
case 'FETCH_GISTS_REQUEST':
    return {...state, fetchMethod: action.fetchMethod, isFetching: true};
    break;
```

Kuva 31. Haun alkamisesta ilmoittaminen ja tilan päivittäminen vastaamaan haun tilannetta.

Haun onnistuessa lähetetään receiveGists-toiminto. ReceiveGists sisältää haun tuloksena saadut gistit, haun päättymisajan (kuva 32). Jos hakutyyppinä on discover, sivutus päivitetään lähettämällä updatePagination-toiminto.

```
//@actions.js
function receiveGists(gists) {
    return {type: 'FETCH_GISTS_SUCCESS', gists, fetchedAt: new Date().getTime() / 1000};
}

//@gists.js
case 'FETCH_GISTS_SUCCESS':
    return {...state, items: action.gists, fetchedAt: action.fetchedAt, isFetching: false};
    break;
```

Kuva 32. Haun onnistumisen käsittely.

Jos haku epäonnistuu, lähetetään gistsFetchFailed-toiminto. GistsFetchFailed ilmoittaa haun päättymisestä sekä välittää haun epäonnistumisen syyn (kuva 33).

```
//@actions.js
function gistsFetchFailed(error) {
    return {type: 'GISTS_FETCH_FAILURE', isFetching: false, error};
}

//@gists.js
case 'GISTS_FETCH_FAILURE':
    return {...state, items: [], fetchError: action.error, isFetching: false};
    break;
```

Kuva 33. Haun epäonnistumisen käsittely.

3.5.3 Aktiivisen gistin hakeminen

Useampaa gistiä haettaessa vastauksena saadut gistit eivät sisällä koodileikkeitä. Gistin sisältämien koodileikkeiden näyttämiseksi gistin tarkemmat tiedot täytyy hakea API:n kautta erikseen. Yksittäinen gist haetaan lähettämällä GET-tyyppinen pyyntö /gists/:id-päätepisteeseen (kuva 34).

Gistin tarkempia tietoja haetaan automaattisesti, kun gist asetetaan aktiiviseksi. Aktiivinen gist voi vaihtua reaktiona käyttäjän klikkaukseen tai listattavien gistien muutokseen (uusi haku tai päivittäminen). Listattavien gistien muuttuessa tulosten ensimmäinen gist asetetaan aktiiviseksi. Käyttäjä voi myös vaihtaa aktiivista gistiä klikkaamalla listaan sijoitettuja li-elementtejä. Klikkauksen kohteena olleen elementin id:tä vastaava gist asetetaan aktiiviseksi ja gistin hakeminen aloitetaan.

```
export function fetchSelectedGist(id) {
    const url = 'https://api.github.com/gists/' + id;

    return (dispatch) => {
        dispatch(requestSelectedGist(id));
        //Tarkistetaan onko gist käyttäjän suosikeissa.
        dispatch(checkIfStarred(id));

        return sendRequest(url, 'GET')
            .then(checkStatus)
            .then(readJson)
            .then((data) => dispatch(receiveSelectedGist(parseSingleGistJson(data))))
            .catch((error) => dispatch(gistFetchFailed(error.message)));
    };
}
```

Kuva 34. Aktiivisen gistin hakeminen.

Aktiivisen gistin hakeminen seuraa pääpiirteittäin listattavien gistien hakuoperaatioita. Haun aikana lähetetään eri toiminnot, mutta haun tilanteesta ilmoittavat toiminnot toimivat käytännössä aivan samalla tavalla sillä erotuksella, että ne päivittävät sovelluksen tilaa aktiivisen gistin osalta. Siinä missä listattavien gistien haku vaikuttaa näkymän vasemman lohkon sisältöön, aktiivisen gistin haku aiheuttaa vastaavat muutokset oikeaan lohkoon.

3.5.4 Gistin lisääminen ja poistaminen suosikeista

Gistin hakemisen yhteydessä selvitetään myös, onko aktiivinen gist jo käyttäjän suosikeissa. Sen selvittämiseksi lähetetään GET-tyyppinen kutsu /gists/:id/star-päätepisteeseen (kuva 35). Vastauksen status-koodin perusteella määritellään aktiivisen gistin isStarred-ominaisuuden arvo. IsStarred-ominaisuuden arvon perusteella käyttäjälle näytetään painike joko gistin suosikkeihin lisäämiseen tai suosikeista poistamiseen.

```
export function checkIfStarred(id) {
    const url = 'https://api.github.com/gists/' + id + '/star';

    return (dispatch) => {
        return sendRequest(url, 'GET')
            .then((response) => {
                if(response.ok) {
                    dispatch(starred());
                //Jos status-koodi on 404, gistiä ei ole asetettu suosikiksi.
                } else if(response.status === 404) {
                    dispatch(notStarred());
                //Jos joku muu virhekoodi, heitetään virhe.
                } else {
                    throw new Error(response.status + ' ' + response.statusText);
                }
            }).catch((error) => dispatch(notify('failure', error.message)));
    };
}
```

Kuva 35. Tarkistetaan onko gist käyttäjän suosikeissa.

Gist lisätään suosikkeihin lähettämällä PUT-tyyppinen pyyntö /gists/:id/star-päätepisteeseen (kuva 36). Suosikeista poistamiseen käytetään samaa päätepistettä, mutta lähetettävä pyyntö on DELETE-tyyppinen (kuva 35).

```
//Lisätään gist suosikkeihin.
export function starGist(id) {
    const url = 'https://api.github.com/gists/' + id + '/star';

    return (dispatch) => {
        dispatch(starring());
        return sendRequest(url, 'PUT', '')
            .then(checkStatus)
            .then(() => {
                dispatch(starred());
                dispatch(notify('success', 'Lisätty suosikkeihin.'));
            }).catch((error) => dispatch(notify(
                'failure',
                'Suosikkeihin lisääminen epäonnistui (' + error.message + ').'
            )));
    };
}

//Poistetaan gist suosikeista.
export function unstarGist(id) {
    const url = 'https://api.github.com/gists/' + id + '/star';

    return (dispatch) => {
        dispatch(starring());
        return sendRequest(url, 'DELETE')
            .then(checkStatus)
            .then(() => {
                dispatch(notStarred());
                dispatch(notify('success', 'Poistettu suosikeista.'));
            }).catch((error) => dispatch(notify(
                'failure',
                'Suosikeista poistaminen epäonnistui (' + error.message + ').'
            )));
    };
}
```

Kuva 36. Gistin lisääminen ja poistaminen suosikeista.

Käyttäjälle näytettävän painikkeen arvoa vaihdetaan Aseta suosikiksi ja Poista suosikeista välillä vaihtamalla aktiivisen gistin isStarred-muuttujan arvoa starred- ja notStarred-toimintojen avulla.

3.5.5 Gistin kopioiminen

Käyttäjä käynnistää kopioimisen painamalla Fork-painiketta. Ennen kopioimista tarkistetaan, että käyttäjä ei ole kopioinut kyseistä gistiä aiemmin tililleen (kuva 37). Gistin kopioinnit haetaan GET-tyyppisellä pyynnöllä /gists/:id/forks-päätepisteestä. Tulokset käydään läpi ja mikäli käyttäjän id löytyy kopioinnin tehneiden käyttäjien listalta, kopiointi keskeytetään. Jos käyttäjä ei ole kopioinut

gistiä aiemmin, kopiointi suoritetaan lähettämällä POST-tyyppinen pyyntö /gists/:id/forks-päätepisteeseen (kuva 38).

```
export function checkIfForked(id) {
    const url = 'https://api.github.com/gists/' + id + '/forks';

    return (dispatch) => {
        return sendRequest(url, 'GET')
            .then(checkStatus)
            .then(readJson)
            .then((data) => {
                let isForked = false;

                //Käydään gistin forkkaukset läpi.
                data.forEach((fork) => {
                    //Jos kirjautuneen käyttäjän id löytyy forkkauksista, estetään forkkaus.
                    if(fork.owner.id === Number(userInfo.user.id)) {
                        dispatch(notify('failure', 'Olet jo forkannut tämän gistin.'));
                        isForked = true;
                    }
                });

                //Forkataan gist jos käyttäjä ei ole jo forkannut gistiä.
                if (!isForked) {
                    dispatch(forkGist(id));
                }
            }).catch((error) => dispatch(notify('failure', error.message)));
    };
}
```

Kuva 37. Tarkistetaan voidaanko gist forkata. (PÄIVITÄ)

```
export function forkGist(id) {
    const url = 'https://api.github.com/gists/' + id + '/forks';

    return (dispatch) => {
        return sendRequest(url, 'POST')
            .then(checkStatus)
            .then(() => dispatch(notify('success', 'Kopioitu tilille.')))
            .catch((error) => dispatch(notify(
                'failure',
                'Kopioiminen epäonnistui (' + error.message + ').'
            )));
    };
}
```

Kuva 37. Gistin kopioiminen.

3.5.6 Gistin poistaminen

Käyttäjä laukaisee gistin poistamisen painamalla Poista-painiketta. Gist poistetaan lähettämällä DELETE-tyyppinen pyyntö /gists/:id-päätepisteeseen (kuva 39).

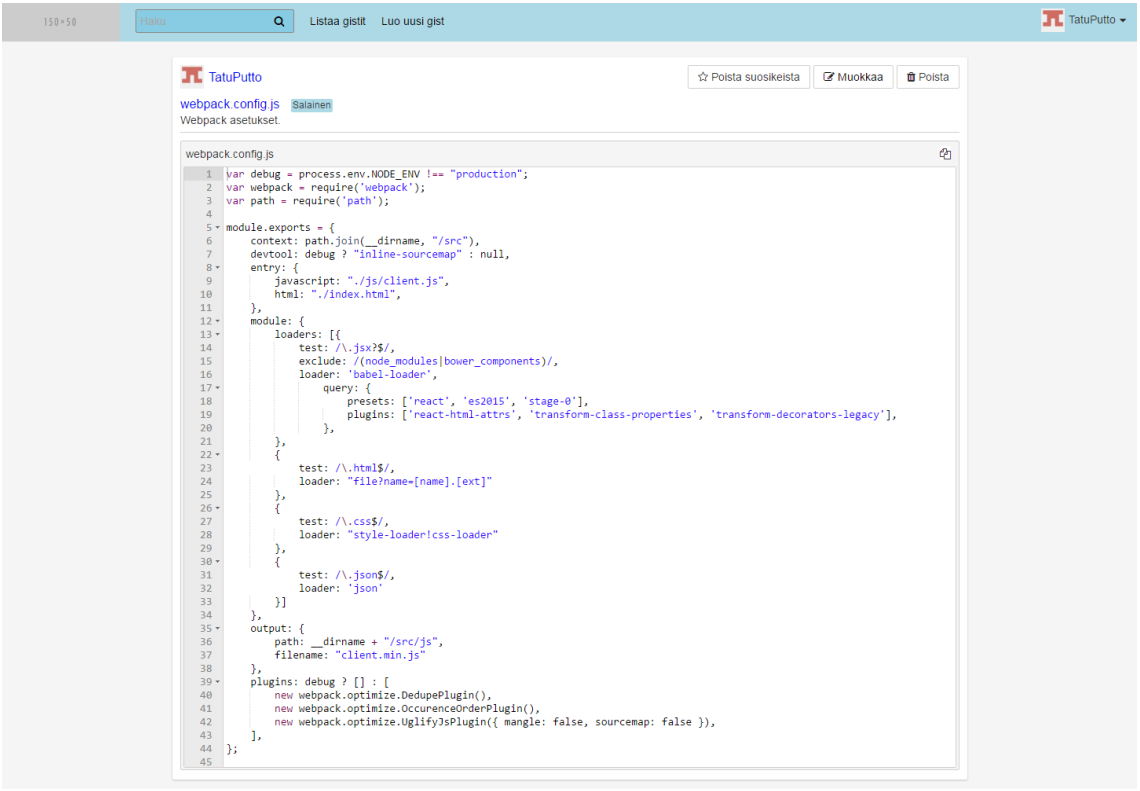
```
export function deleteGist(id) {
    const url = 'https://api.github.com/gists/' + id;

    return (dispatch) => {
        return sendRequest(url, 'DELETE')
            .then(checkStatus)
            .then(() => {
                dispatch(invalidateGist());
                dispatch(removeGistFromList(id));

                //Jos poistetaan yksittäisen gistin näkymässä,
                //ohjataan käyttäjä takaisin listausnäkymään.
                if(location.pathname === '/opinnaytetyo/gist/' + id) {
                    browserHistory.push('/opinnaytetyo');
                }
            }).catch((error) => dispatch(notify(
                'failure',
                'Poistaminen epäonnistui (' + error.message + ').'
            )));
    };
}
```

Kuva 39. Gistin poistaminen Gist-palvelusta.

Jos poistaminen onnistuu, aktiivinen gist poistetaan näkymästä ja id:tä vastaava li-elementti poistetaan listasta lähettämällä invalidateGist- ja removeGistFromList-toiminnot. Jos poistaminen suoritetaan yksittäisen gistin näkymästä, käyttäjä ohjataan takaisin listausnäkymään.

3.6 Yksittäisen gistin näkymä

Yksittäisen gistin tarkastelua varten luotiin myös näkymä (kuva 40). Näkymän päätarkoituksena gistin tarkastelun ohella on antaa konkreettinen URL-osoite yksittäiselle gistille, jotta kyseinen resurssi voidaan helposti esimerkiksi linkittää tai lisätä kirjanmerkiksi.



Kuva 40. Yksittäisen gistin näkymä.

Näkymän toiminnallisuus uudelleen käytetään suoraan listausnäkymästä. Toiminnallisuuden lisäksi myös näkymän rakenne muodostuu lähes kokonaan listausnäkymästä uudelleen käytettävistä komponenteista. Näkymälle luotiin SingleGist-komponentti, joka yhdistää näkymän Reduxiin ja muodostaa näkymän listausnäkymästä uudelleenkäytettävistä komponenteista (ShowActiveGist-komponentti kokonaisuus).

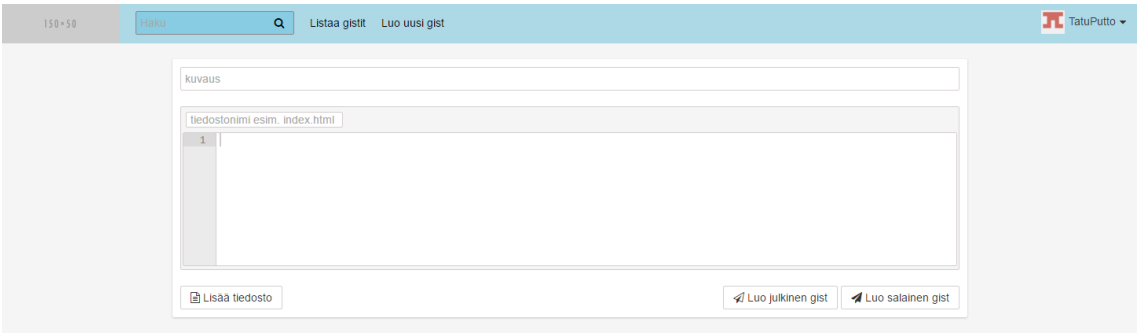
Gist haetaan käyttäjän saapuessa näkymään. Haun käynnistää fetchSelectedGistOnEnter-funktio, jota kutsutaan kun käyttäjä saapuu yksittäisen gistin näkymään (kuva 41). Ennen gistin hakemista tarkistetaan, onko gistiä tarve hakea. Jos pyyntöä vastaava gist on jo tallennettuna välimuistiin, käytetään välimuistiin tallennettua gistiä uudestaan hakemisen sijaan. Jos gist joudutaan hakemaan, käytetään luvussa 3.3.2 esiteltyä yksittäisen gistin hakuprosessia.

```
export function fetchSelectedGistOnEnter(nextState) {
    let gistId = nextState.params.gistId;
    let state = store.getState();

    //Haetaan gist, jos tilaan ei ole tallennettu gistiä
    //tai käyttäjän pyytämä gist ei vastaa tilaan tallennettua gist.
    if(!state.activeGist.gist.hasOwnProperty('id') || state.activeGist.gistId !== gistId) {
        return store.dispatch(fetchSelectedGist(gistId));
    }
}
```

Kuva 41. Määritellään täytyykö gist hakea näkymään saavuttaessa vai voidaanko käyttää välimuistiin tallennettua gistiä.

3.7 Gistin luontinäkymä

Gistien luomisesta pyrittiin tekemään mahdollisimman yksinkertaista, joten näkymästä tehtiin hyvin pelkistetty (kuva 42). Näkymään lisättiin vain oleelliset elementit: tekstikenttä kuvaukselle, tiedostokentät sekä painikkeet tiedostokenttien lisäämiseen ja gistin luomiseen julkisena tai salaisena.

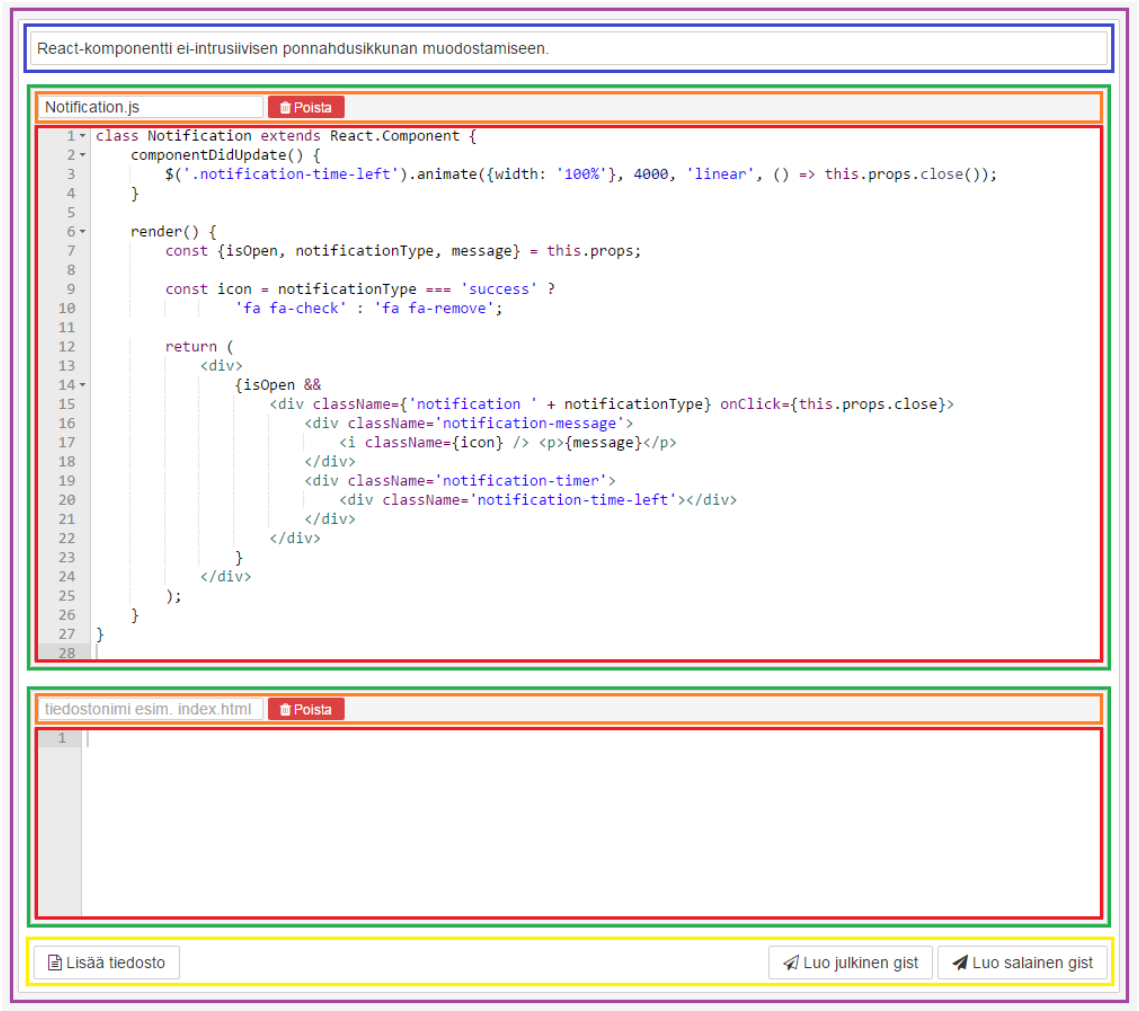


Kuva 42. Luontinäkymä.

Luontinäkymä jaetaan kuvan 43 mukaisesti komponentteihin:

- CreateGist (violetti): koostaa komponentit näkymäksi ja hoitaa näkymän käyttöliittymän toimintaan liittyvän logiikan.
- Tekstikenttä (sininen): tekstikenttä gistin kuvausta varten.
- GistFile (vihreä): muodostaa tiedostokentän FileInfoWithInput- ja Editor-komponenteista. GistFile-komponentista luodaan yksi ilmentymä jokaista tiedostokenttää kohden.
- FileInfoWithInput (oranssi): tekstikenttä tiedostonimelle ja tarvittaessa painike tiedostokentän poistamiseen.

- Editor (punainen): muodostaa div-elementin, ja luo elementtiin ilmentymän ACE:sta.
- Painikkeet (keltainen): painikkeet tiedostokenttien lisäämiseen ja gistin luomiseen salaisena tai julkisena.



Kuva 43. Luontinäkymän komponenttihierarkia.

Tiedostokenttien renderöinti rakennettiin CreateGist-komponentin tiladatan päivittymisen ympärille. CreateGist säilyttää tiladatanaan kulloinkin näkymään renderöitävien tiedostokenttien id:t editors-taulukossa sekä luotujen tiedostokenttien määrän editorsCreated-muuttujassa. CreateGist renderöi tiedostokentät editors-taulukon arvojen pohjalta (kuva 44).

```
//Luodaan jokaista tilaan tallennettua editori id:tä kohden yksi tiedostokenttä.
const fileFields = this.state.editors.map((editorId) => {
    return (
        <GistFile
            key={editorId}
            isRemovable={isRemovable}
            remove={this.removeFile}
            editorId={editorId}
            isReadOnly={false}
        />
    );
}, this);

return (
    <div className='create'>
        <input type='text' className='description' placeholder='Kuvaus' />

        <div className='files'>
            {fileFields}
        </div>
```

Kuva 44. Tiedostokenttien luominen CreateGist-komponentin tilan pohjalta.

Tiedostokenttien lisäämiseen ja poistamiseen luotiin addFile- ja removeFile-metodit. Metodit eivät lisää tai poista kenttiä suoraan DOM:sta, vaan muuttavat CreateGist-komponentin tilaa (kuva 45). Lisättävän tiedostokentän id säilytetään aina uniikkina muodostamalla kentän id editorsCreated-muuttujan arvon perusteella. Näin varmistutaan siitä, että oikea kenttä poistetaan aina vaikka käyttäjä poistaisi kenttiä satunnaisessa järjestyksessä.

```
//Lisätään uusi tiedostokenttä.
addFile() {
    this.setState({
        editors: this.state.editors.concat(['editor' + this.state.editorsCreated]),
        editorsCreated: this.state.editorsCreated + 1,
    });
}

//Poistetaan valittu tiedostokenttä.
removeFile(id) {
    if(confirm('Haluatko varmasti poistaa tämän kentän?')) {
        let editors = this.state.editors;
        editors.splice(editors.indexOf(id), 1);

        this.setState({editors});
    }
}
```

Kuva 45. Tiedostokenttien lisäämisen ja poistaminen.

Luotavan gistin sisältö esitetään JSON-oliona. Olioon sisällytetään kuvaus, yksityisyys aste (julkinen tai salainen) ja tiedostot. Jotta gist saadaan luotua onnistuneesti Gist-palveluun, täytyy muodostetun olion seurata Gist API:n määrittämää rakennetta (kuva 46). Jos olion rakenne on virheellinen, luontipyyntö epäonnistuu. (GitHub 2016b.)

Name	Type	Description
`files`	`object`	**Required.** Files that make up this gist.
`description`	`string`	A description of the gist.
`public`	`boolean`	Indicates whether the gist is public. Default: `false`

```
{
  "description": "the description for this gist",
  "public": true,
  "files": {
    "file1.txt": {
      "content": "String file contents"
    }
  }
}
```

Kuva 46. Luotavaa gistiä kuvaavan JSON-olion rakenne (GitHub 2016b).

Käyttäjä käynnistää luontiprosessin Luo julkinen gist- tai Luo salainen gist-painiketta painamalla. Luotavan gistin tiedot kerätään DOM:sta, tiedot koostetaan olioksi ja koostettu olio lähetetään JSON-muodossa eteenpäin luontipyynnön lähettävälle createGist-toiminnon käsittelijälle (kuva 47).

```
//Koostetaan tiedot yhdeksi olioksi ja lähetetään se eteenpäin.
getGistInfo(isPublic) {
    let gist = {};
    let files = {};
    const description = $('.description').val();
    const fileFields = $('.gist-file');
    const editors = this.state.editors;

    //Kerätään tiedostonimet ja lähdekoodit tiedostokentistä.
    for(let i = 0; i < fileFields.length; i++) {
        const filename = $(fileFields[i]).find('input:text').val();
        const content = ace.edit(editors[i]).getValue();

        const file = {filename, content};
        files[filename] = file;
    }

    //Koostetaan olio.
    gist['description'] = description;
    gist['ispublic'] = isPublic;
    gist['files'] = files;

    //Lähetetään koostettu olio JSON-muodossa eteenpäin.
    this.props.create(JSON.stringify(gist));
}
```

Kuva 47. GetGistInfo-metodi.

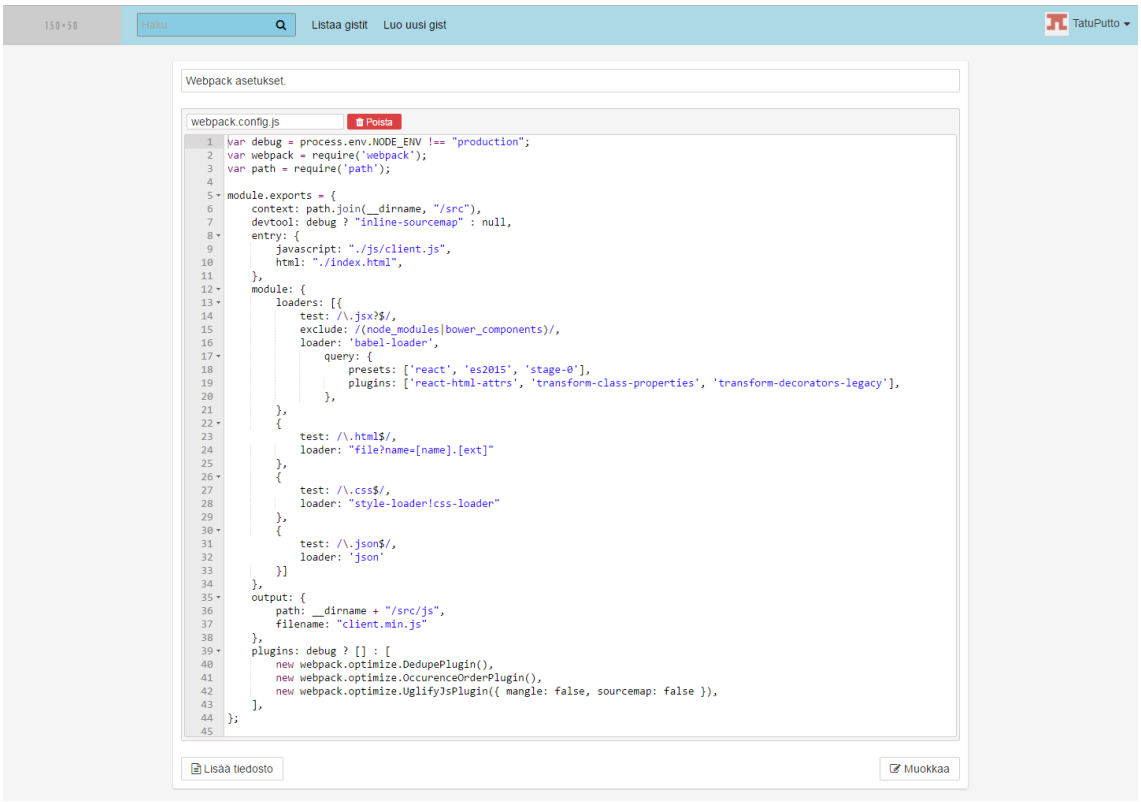
Gist luodaan lähettämällä POST-tyyppinen pyyntö /gists-päätepisteeseen (kuva 48). Pyynnön sisällöksi asetetaan gistin kuvaava JSON-olio. Mikäli gistin luominen onnistuu, käyttäjä ohjataan luodun gistin näkymään. Jos luominen epäonnistuu, käyttäjälle ilmoitetaan pyynnön epäonnistumisen syy.

```
export function createGist(gistJson) {
    const url = 'https://api.github.com/gists';

    return (dispatch) => {
        return sendRequest(url, 'POST', gistJson)
            .then(checkStatus)
            .then(readJson)
            .then((data) => {
                //Asetetaan muokattu gist aktiiviseksi
                //ja ohjataan käyttäjä luodun gistin näkymään.
                dispatch(receiveSelectedGist(parseSingleGistJson(data)));
                dispatch(notify('success', 'Gistin luominen onnistui.'));
                browserHistory.push('/opinnaytetyo/gist/' + data.id);
            }).catch((error) => dispatch(notify(
                'failure',
                'Gistin luominen epäonnistui (' + error.message + ').'
            )));
    };
}
```

Kuva 48. Gistin luontipyynnön lähettäminen ja vastauksen käsittely.

3.8 Gistin muokkausnäkymä

Gistien muokkaamista varten rakennettiin oma näkymänsä (kuva 49). Rakenteeltaan näkymä oli lähes identtinen luontinäkymän kanssa, joten esitykselliset komponentit voitiin uudelleen käyttää sellaisenaan. Näkymän koostamiseen ja logiikan hoitamiseen määriteltiin EditGist-komponentti ja Luo julkinen gist- ja Luo salainen gist-painikkeet korvattiin Päivitä-painikkeella.



Kuva 49. Gistin muokkausnäkymä.

Gistin muokkaamiseksi gist on ensin haettava. Hakulogiikka uudelleen käytettiin yksittäisen gistin näkymästä. Jotta gistin tiedostot saadaan esitettyä näkymässä, tiedostot tallennetaan EditGist-komponentin tilatiedoksi (kuva 50). Tiedostojen pohjalta muodostettiin files-taulukko, jonka arvojen perusteella tiedostokentät renderöidään (kuva 50). Files-taulukkoon tallennetaan tiedostonimi, koodileike, aktiivisuus (näytetäänkö kyseinen tiedosto käyttöliittymässä) ja onko tiedosto alkuperäinen vai muokkauksen aikana lisätty. Tiedostot tallennetaan myös originalFiles-taulukkoon, jonka arvoja käytetään alkuperäisiin tiedostoihin tehtyjen muutosten löytämiseksi.

```
//Gist on ladattu välimuistista näkymään saavuttaessa.
componentDidMount() {
    if(this.props.gist.hasOwnProperty('id')) {
        this.initializeFiles(this.props.gist.files);
    }
}

//Gist täytyy hakea -> viivytetään tiedostokenttien alustamista, kunnes haku on valmis.
componentWillReceiveProps(nextProps) {
    if(nextProps.gist.hasOwnProperty('id')) {
        this.initializeFiles(nextProps.gist.files);
    }
}

initializeFiles(files) {
    //Tallennetaan gistin tiedot renderöintiin käytettävään taulukkoon
    for(let i = 0; i < files.length; i++) {
        files[i].editorId = 'editor' + i;
        files[i].isActive = true;
        files[i].isOriginal = true;
    }

    //Tallennetaan tiedostot myös erilliseen taulukkoon
    //muokkausvaiheessa tehtävää vertailua varten.
    this.setState({files, originalFiles: files, editorsCreated: files.length});
}
```

Kuva 50. EditGist-komponentin tiladatan asettaminen.

```
//Luodaan tiedostokentät, jotka ovat aktiivisia.
const fileFields = this.state.files.map((file) => {
    if(file.isActive) {
        return (
            <GistFile
                key={file.editorId}
                filename={file.filename}
                isRemovable={true}
                remove={this.removeFile}
                onChange={this.handleOnChange}
                editorId={file.editorId}
                isReadOnly={false}
                value={file.content}
            />
        );
    }
}, this);

return (
    <div className='edit'>
        <input type='text' className='description'
                placeholder='Kuvaus' defaultValue={gist.description} />

        <div className='files'>
            {fileFields}
        </div>
```

Kuva 51. Osa EditGist-komponentin render-metodista.

EditGist kuvaa tiedostokentät files-taulukon arvojen perusteella. Renderöitävät tiedostokentät määritellään isActive-ominaisuudella. Kun tiedostokenttä poistetaan näkymästä (kuva 52), oliota ei poisteta files-taulukosta, vaan sen isActive-ominaisuuden arvo asetetaan epätodeksi. Näin varmistutaan, että muokkausvaiheessa voidaan helposti huomata jos alkuperäinen tiedosto on poistettu.

```
//Lisätään tiedostokenttä.
addFile() {
    this.setState({
        files: this.state.files.concat({
            filename: '',
            content: '',
            editorId: 'editor' + this.state.editorsCreated,
            isActive: true,
            isOriginal: false,
        }),
        editorsCreated: this.state.editorsCreated + 1,
    });
}

//Poistetaan tiedostokenttä.
removeFile(id) {
    if(confirm('Haluatko varmasti poistaa tämän kentän?')) {
        let files = this.state.files;

        for(let i = 0; i < files.length; i++) {
            if(files[i].editorId === id) {
                files[i].isActive = false;
            }
        }
        this.setState({files});
    }
}
```

Kuva 52. Tiedostokenttien lisääminen ja poistaminen.

Gistiin tehtävät muutokset kuvataan JSON-oliolla. Olioon sisällytetään kuvaus, muutoksia sisältävät tiedostot ja muokkauksen yhteydessä lisätyt uudet tiedostot. Gistin alkuperäiset tiedostot, joita ei mainita oliossa, kantautuvat automaattisesti gistin päivitettyyn versioon. Olion täytyy vastata Gist API:n määrittelemää rakennetta. Jos olion rakenne ei vastaa kuvan 53 esittämää rakennetta, pyyntö epäonnistuu. (GitHub 2016b.)

Name	Type	Description
description	string	A description of the gist.
files	object	Files that make up this gist.
content	string	Updated file contents.
filename	string	New name for this file.

```
{
  "description": "the description for this gist",
  "files": {
    "file1.txt": {
      "content": "updated file contents"
    },
    "old_name.txt": {
      "filename": "new_name.txt",
      "content": "modified contents"
    },
    "new_file.txt": {
      "content": "a new file"
    },
    "delete_this_file.txt": null
  }
}
```

Kuva 53. Gistin muutokset kuvaavan JSON-olion rakenne (GitHub 2016b).

Käyttäjä käynnistää gistin muokkausprosessin painamalla Muokkaa-painiketta. Muokkausprosessi aloitetaan määrittelemällä JSON-olioon sisällytettävät tiedostot. Sisällytettävät tiedostot määritellään käymällä files-taulukon tiedostot yksi kerrallaan läpi (kuva 54). Määrittely suoritetaan tiedoston isActive- ja isOriginal-muuttujien arvojen perusteella, isActive kuvaa onko tiedosto poistettu ja isOriginal kuvaa onko tiedosto alkuperäinen. Muuttujien arvojen perusteella selvitetään, onko tiedosto muokkausistunnon aikana lisätty vai gistin alkuperäinen tiedosto ja onko tiedosto poistettu vai vielä aktiivisena DOM:ssa. Muokkausistunnon aikana luodut tiedostot, jotka ovat aktiivisia, lisätään suoraan olioon. Jos tiedosto on alkuperäinen ja aktiivinen, tiedosto tarkistetaan muutosten varalta. Jos muutoksia löytyy, tiedosto sisällytetään olioon ja arvoksi annetaan muuttuneet tiedot. Jos tiedosto on alkuperäinen ja poistettu istunnon aikana, tiedosto sisällytetään olioon ja merkitään poistettavaksi antamalla arvoksi null.

```
//Haetaan tiedostonimet DOM:sta.
let filenames = document.getElementsByClassName('filename');
let modifiedFiles = {};
let offset = 0;

for(let i = 0; i < files.length; i++) {
    //Uusi tiedostokenttä on luotu ja poistettu ennen muokkausta.
    if(!files[i].isOriginal && !files[i].isActive) {
        offset++;
    //Jos tiedosto on uusi, lisätään suoraan.
    } else if(!files[i].isOriginal) {
        modifiedFiles[filenames[(i - offset)].value] = {
            filename: filenames[(i - offset)].value,
            content: ace.edit(files[i].editorId).getValue(),
        };
    //Jos tiedosto on alkuperäinen ja poistettu.
    } else if(files[i].isOriginal && !files[i].isActive) {
        modifiedFiles[files[i].filename] = null;
        offset++;
    //Jos tiedosto on alkuperäinen, tarkistetaan onko siihen tehty muutoksia.
    } else {
        // Alkuperäinen tiedostonimi ja koodileike.
        const originalFilename = originalFiles[i].filename;
        const originalContent = originalFiles[i].content;

        // Tiedostonimi ja koodileike muokkauksen jälkeen.
        const filenameOnUpdate = filenames[(i - offset)].value;
        const contentOnUpdate = ace.edit(files[i].editorId).getValue();

        const nameChanged = originalFilename !== filenameOnUpdate ? true : false;
        const contentChanged = originalContent !== contentOnUpdate ? true : false;

        //Riippuen muutoksista, lisätään päivitetty tiedostonimi ja/tai koodileike.
        if(nameChanged && contentChanged) {
            modifiedFiles[originalFilename] = {
                filename: filenameOnUpdate,
                content: contentOnUpdate,
            };
        } else if(nameChanged) {
            modifiedFiles[originalFilename] = {filename: filenameOnUpdate};
        } else if(contentChanged) {
            modifiedFiles[originalFilename] = {content: contentOnUpdate};
        }
    }
}
```

Kuva 54. Muokkauspyyntöön sisällytettävien tiedostojen määrittely.

Gistin muokkaaminen suoritetaan lähettämällä PATCH-tyyppinen pyyntö /gists/:id-päätepisteeseen. Pyynnön sisällöksi asetetaan muutokset kuvaava JSON-olio. Jos muokkaaminen onnistuu, muokattu gist asetetaan aktiiviseksi ja käyttäjä ohjataan muokatun gistin näkymään. Jos muokkaaminen ei onnistu, pyynnön epäonnistumisen syy ilmoitetaan käyttäjälle (kuva 55).

```
export function editGist(id, editJson) {
    const url = 'https://api.github.com/gists/' + id;

    return (dispatch) => {
        return sendRequest(url, 'PATCH', editJson)
            .then(checkStatus)
            .then(readJson)
            .then((data) => {
                dispatch(notify('success', 'Gistin muokkaaminen onnistui.'));
                //Asetetaan muokattu gist aktiiviseksi
                //ja ohjataan käyttäjä muokatun gistin näkymään.
                dispatch(receiveSelectedGist(parseSingleGistJson(data)));
                browserHistory.push('/opinnaytetyo/gist/' + data.id);
            }).catch((error) => dispatch(notify(
                'failure',
                'Gistin muokkaaminen ei onnistunut (' + error.message + ').'
            )));
    };
}
```

Kuva 55. Gistin muokkauspyynnön lähettäminen ja vastauksen käsittely.

4 Tulokset

Yksi opinnäytetyön keskeisimmistä tavoitteista oli luoda sovellus, joka käyttää pilvipohjaista tiedon persistointimetodia perinteisen tietokantaratkaisun sijaan. Projektin alussa en ollut ollenkaan varma siitä, toimisiko tällainen toteutus käytännössä. Alkuvaikeuksien jälkeen ulkoisen API:n käyttö alkoi muuttua helpommaksi ja ajan mittaan sen käyttämisestä tuli täysin luonnollinen osa sovellusta. Käytännössä eroa ei luultavasti huomattaisi, jos ulkoinen API korvattaisiin palvelinsovelluksen yhteyteen toteutetulla tietokannalla. Ulkoinen API persistointimetodina oli siis onnistunut valinta ja sitä onnistuttiin hyödyntämään tehokkaasti sovelluksen datan persistoitiin. Datan persistoinnin ulkoistaminen karsi myös huomattavasti palvelinsovelluksen toteutuksen monimutkaisuutta, minkä ansiosta pystyin keskittymään ammatillisen kehittymiseni kannalta olennaisimpiin tehtäviin.

Pilvipohjaisen persistointimetodin ohella sovelluksen tuli hyödyntää SPA- sekä Redux-arkkitehtuureita. SPA-arkkitehtuuri toteutettiin onnistuneesti, mutta hyödyt jäivät vähäisemmiksi kuin alussa kuviteltiin. SPA-arkkitehtuurin yleiset ongelmat onnistuttiin kuitenkin käytännössä välttämään kokonaan, joten se tuo sovellukseen lisäarvoa.

Reduxin vaikutukset näkyivät sovelluksen toiminnassa ja kehitystyössä SPA-arkkitehtuuria huomattavasti enemmän. Redux-arkkitehtuuri teki sovelluksesta erittäin varmatoimisen. Sovellus nojaa hyvin raskaasti asynkronisuuteen. Monta asynkronista toimintoa tapahtuu usein päällekkäin. Tulosten ja virhetilanteiden lisäksi jokainen asynkronisista operaatioista vaikuttaa myös käyttöliittymän tilaan. Ilman Reduxia tämä toteutus olisi ollut todella haastavaa saada aikaiseksi toimintavarmalla tavalla. Monimutkaisuutta olisi jouduttu karsimaan käytettävyyden kustannuksella. Kehitystyössä Reduxin selkeä toimintamalli toi yhtenäisyyttä koodikantaan. Uusia toiminnallisuuksia oli helppo lisätä, kun kaikki toiminnot noudattavat samoja vaiheita. Tämä nopeuttaa varmasti myös sovelluksen jatkokehitystä tulevaisuudessa.

Arkkitehtuuristen tavoitteiden lisäksi määriteltiin, mitä toiminnallisuuksia sovelluksen tulisi sisältää. Toiminnallisuudet olivat luominen, lukeminen, muokkaaminen, poistaminen, suosikkeihin lisääminen ja suosikeista poistaminen. Lisäksi muiden käyttäjien Gist-palveluun luomia koodileikkeitä täytyi pystyä lukemaan, lisäämään ja poistamaan suosikeista ja kopioimaan omalle tilille. Kaikki edellä mainitut toiminnallisuudet saatiin toteutettua. Toiminnallisuuksia toteutettaessa erityistä huomiota kiinnitettiin varmatoimisuuteen ja siihen, että toiminnolla on odotettu lopputulos (toimii samalla tavalla kuin Gist-palvelun vastaava toiminto). Lisäksi varmistettiin, että toiminnot ovat helppokäyttöisiä ja ne toimivat mielekkäästi osana käyttöliittymää.

Arkkitehtuuriset valinnat ja toteutukset toimivat hyvin yhdessä ja tukevat niin sovelluksen käytettävyyttä kuin toimintaakin. Tarvittavat toiminnot implementoitiin onnistuneesti. Ne ovat varmatoimisia, helppokäyttöisiä, toimivat odotetusti ja sulautuvat käyttöliittymään mielekkäällä tavalla. Tältä pohjalta voidaan todeta, että ensisijaisen tavoite täyttyi hyvin.

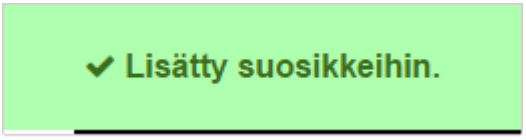
Toissijaisena tavoitteena oli tuottaa käyttöliittymä, joka parantaisi Gist-palvelun käytettävyyttä. Tätä tavoitetta ei pidetty realistisena Gist API:n tarjoamien toiminnallisuuksien sekä aikataulun rajallisuuden takia. Sovellusta lähdettiin kuitenkin toteuttamaan tämä tavoite mielessä pitäen. Ulkoasuun ja käytettävyyteen kiinnitettiin paljon huomiota. Käyttöliittymästä saatiinkin hyvin tyylikäs ja käytettävyydeltään hyvä kokonaisuus. Ei voida kuitenkaan väittää, että sovellus parantaisi Gist-palvelun käytettävyyttä merkittävällä tavalla.

Koska opinnäytetyön lopputulos on vaihtoehtoinen käyttöliittymä Gist-palvelulle, käyttöliittymän käytettävyyttä päätettiin arvioida tarkemmin. Arvioinnin tuloksia tarkastellessa täytyy ottaa huomioon, että arvioinnin on tehnyt vain yksi henkilö ja että se on tehty kehittäjän näkökulmasta. Arvio on siis hyvin subjektiivinen, joten tuloksiin tulee suhtautua varauksella. Loppukäyttäjän käyttökokemus ei välttämättä vastaa arvion pohjalta esitettyjä tuloksia.

Arvioinnissa käytettiin apuna Johan Nielsenin laatimaa heuristisen arvioinnin muistilistaa. Lista koostuu 10 kohdasta, joiden pohjalta käyttöliittymää tarkastellaan: (Nielsen 1995.)

1. Palvelun tilan näkyvyys

”Käyttäjän pitäisi aina pystyä nopeasti huomaamaan mikä on palvelun tila ja käyttäjän sijainti palvelussa” (Aalto-yliopisto 2016). Palvelun tilan näkyvyyteen jouduttiin kiinnittämään erityistä huomiota. SPA-arkkitehtuuri yhdistettynä lähes täysin asynkroniseen datan hakumalliin aiheuttaa käyttäjälle hyvin helposti hämmennystä sovelluksen tilasta. Tilan muutoksesta indikoimiseen käytettiin mm. latausindikaattoreita ja visuaalisia tehokeinoja. Esimerkiksi käynnissä olevasta latauksesta ilmoitetaan latausindikaattorilla tai elementin opasiteettia laskemalla. Toiminnot, jotka eivät muuta esitettävää sisältöä merkittävästi (suosikkeihin lisääminen, kopioiminen, yms.), ilmoittavat tuloksesta Notifications-komponentin välityksellä (kuva 56).



Kuva 56. Ilmoittaminen Notifications-komponentin välityksellä.

Käyttäjän sijainnin esittäminen riittävällä tasolla on toinen SPA-arkkitehtuuria noudattavan sovelluksen ongelmista. React Routerin avulla selaimen osoitepalkissa näytettävä URL-osoite saatiin kuitenkin pidettyä hyvin synkronoituna näkymien vaihdoksien välillä, jotta käyttäjä näkee aina selkeästi missä näkymässä hän kulloinkin on. Sijainnin hahmottamista voitaisiin parantaa esimerkiksi lisäämällä navigointipolku (breadcrumbs).

2. Palvelun ja tosielämän vastaavuus

”Palvelun pitäisi käyttää tavallisesta elämästä tuttuja termejä, sanontoja ja käsitteitä mieluummin kuin palvelun omaa erikoistermistöä” (Aalto-yliopisto 2016). Toiminnallisuudet, ilmoitukset yms. on pyritty pitämään mahdollisimman selkokielisinä. Ohjelmointiin liittyviä alan ammattitermejä käytetään kuitenkin jossain määrin. Termejä valittaessa otettiin huomioon, että sovelluksen pääasiallinen käyttäjäkunta on ohjelmoijat.

3. Käyttäjän kontrolli ja vapaus

> Käyttäjän pitäisi päästä nopeasti ja vaivatta takaisin kunkin vaiheen alkutilaan, tehtyään epätoivotun tai virheellisen valinnan. "Peru" ja "Tee uudestaan" toiminnot ovat suositeltavia. Palvelu ei myöskään saisi tehdä häiritseviä asioita käyttäjän tahtoa vasten tai tältä kysymättä. (Aalto-yliopisto 2016.)

Käyttäjä pääsee vaivattomasti pois näkymästä tai tilanteesta selaimen takaisin- ja eteenpäin-painikkeiden avulla. Lisäksi navigaatio sijaitsee ylätunnisteessa, jonka kautta käyttäjä pääsee aina helposti takaisin alkunäkymään.

Muutoksien perumista ja uudelleen tekemistä tuetaan editoreissa ja tekstikentissä. Käyttäjä voi hyödyntää historiaa ctrl + z ja ctrl + y -näppäinyhdistelmillä. Toisaalta käyttäjä voi esimerkiksi luontinäkymässä poistaa tiedostokentän vahingossa, eikä kenttää pystytä palauttamaan. Vahinkotilanteita

pyritään välttämään kysymällä aina käyttäjän lupa ennen mahdollisesti haitallisen toiminnon suorittamista.

4. Jatkuvuus ja standardit

> Viestien ja toimintojen pitäisi tarkoittaa yhteneväisesti aina samoja asioita (sanoja tai merkityksiä ei saisi vaihtaa lennossa). Olemassa olevia verkko- ja muita standardeja pitäisi hyödyntää yhteneväisyyteen pyrittäessä. (Aalto-yliopisto 2016).

Elementit ja toiminnot toimivat ja näyttävät pääsääntöisesti samalta läpi sovelluksen. Poikkeuksena ovat editorit, joiden toiminta vaihtelee näkymien välillä. Listausnäkymässä sekä yksittäisen gistin näkymässä editorit asetetaan lukusuojattuun tilaan, kun taas luontinäkymässä ja muokkausnäkymässä editorien sisältöä voidaan muuttaa vapaasti.

Sovelluksesta pyrittiin tekemään Gist-palvelusta selvästi erillinen kokonaisuus. Toiminnallisuuksien tuli kuitenkin toimia suurin piirtein samalla tavalla kuin Gist-palvelussa. Sovelluksen käyttämisen tulisi olla helppoa aikaisemmin Gist-palvelua käyttäneelle käyttäjälle.

5. Virheiden ehkäisy

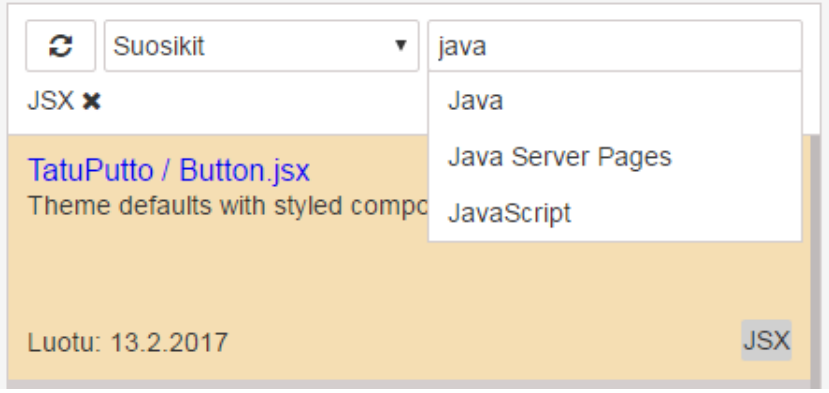
> Palvelun pitäisi tunnistaa mahdolliset virhetilanteet ja estää niiden toistuminen kertomalla käyttäjälle ennen virheen tapahtumista. Opastus pitäisi olla aina helposti saatavilla ja ymmärrettävissä. (Aalto-yliopisto 2016.)

Virheiden ehkäisyssä on paljon eroja näkymien välillä. Listausnäkymä ja yksittäisen gistin näkymä ovat hyvin toimintavarmoja eli virhetilanteita on vaikea saada aikaan. Luonti- ja muokkausnäkymät ovat huomattavasti virhealttiimpia, lähinnä validoinnin puutteen takia. Validoinnin avulla voitaisiin välttyä turhilta virhetilanteilta. Esimerkiksi ei hyväksytyistä erikoismerkeistä tulisi huomauttaa tai merkit tulisi korvata hyväksyttävillä vastineilla. Tämän kohdan kannalta on erityisen tärkeää myös, että käyttäjältä kysytään lupa ennen mahdollisesti haitallisen toiminnon suorittamista.

6. Tunnistaminen mieluummin kuin muistaminen

> Asioiden, toimintojen ja vaihtoehtojen pitäisi olla näkyvissä käyttöliittymässä. Käyttöliittymän painikkeiden ja syötteiden pitäisi liittyä palvelun toimintoihin loogisesti, niin että näiden vastaavuus on pääteltävissä helposti. Käyttäjää ei saisi pakottaa muistamaan asioita ruudulta toiselle siirryttäessä. (Aalto-yliopisto 2016.)

Muistikuormituksen vähentämiseen jouduttiin kiinnittämään huomiota varsinkin listausnäkymässä. Listausnäkymässä on paljon liikkuvia osia: hakutyyppi, aktiiviset suodattimet, suodatusvaihtoehdot, aktiivinen gist jne. Muistikuormitusta pyrittiin vähentämään esittämällä asiat, toiminnot ja vaihtoehdot elementtien arvoina tai omina elementteinään tai elementin visuaalista ilmettä muuttamalla (kuva 57).



Kuva 57. Muistikuormituksen vähentäminen.

7. Käytön tehokkuus ja joustavuus

”Käyttöliittymän tulisi tarjota kokeneelle käyttäjälle oikopolkuja, vaikuttamatta silti aloittelevan käyttäjän käyttökokemukseen” (Aalto-yliopisto 2016). Käyttöä tehostetaan pikanäppäinten avulla. Esimerkiksi listattavia gistejä voidaan selata nuolinäppäimillä ja aktiivisen gistin muokkaamiseen voidaan siirtyä shift + e -näppäinyhdistelmää painamalla. Pääsääntöisesti toiminnot määriteltiin näppäinyhdistelmien taakse, jotta käyttäjä ei laukaise niitä vahingossa.

8. Esteettinen ja minimalistinen suunnittelu

”Ruudulla pitäisi olla ne elementit, jotka ilmaisevat halutun tiedon, toiminnot, tunnelman ja tyylin, ei enempää. Ilmaisun ei pitäisi olla vaikeasti ymmärrettävää (ellei se ole palvelun kantava idea).” (Aalto-yliopisto 2016.) Tässä onnistuttiin näkymien välillä vaihtelevasti. Yksittäisen gistin näkymässä, luontinäkymässä sekä muokkausnäkymässä turha informaation on saatu karsittua käytännössä kokonaan. Listausnäkymässä informaatiota on todella paljon, mutta se on väistämätöntä valitulla asettelulla.

9. Virhetilanteiden tunnistaminen, ilmoittaminen ja korjaaminen

”Virheilmoitusten pitäisi kertoa selkokielellä mitä tapahtui, miksi näin kävi, miten asia voidaan korjata ja kuinka se voidaan välttää ensi kerralla.” (Aalto-yliopisto 2016). Virheilmoituksessa kerrotaan selkokielellä mitä tapahtui (kuva 58). Toisaalta virheilmoituksen perään lisättävä varsinainen virheen syy ilmoitetaan selkokielisen viestin sijaan status-koodilla.

Kuva 58. Virheestä ilmoittaminen.

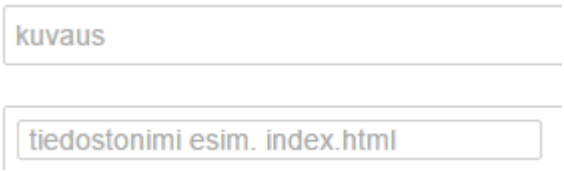
Käyttäjälle ei anneta minkäänlaisia ohjeita virhetilanteen korjaamiseksi. Tämä olisi tärkeä korjata varsinkin gistin luomiseen ja muokkaamiseen liittyvissä ongelmissa, joissa virhetilanteet voivat olla hyvin epäselviä käyttäjälle.

10. Opastus ja ohjeistus

”Vaikka käytön pitäisi tapahtua ilman opastusta ja ohjeita, ovat ne usein välttämättömiä käyttäjille. Näiden pitäisi olla helposti saatavilla, nopeasti etsittävissä, toimintaan ohjaavia, käyttötilannetta tukevia ja riittävän lyhyitä.” (Aalto-yliopisto 2016.)

Käyttämisen katsottiin olevan sen verran selkeää, että erillistä ohjeistusta esimerkiksi opastenäkymän muodossa ei katsottu tarpeelliseksi. Sen sijaan

käyttäjää ohjeistetaan käyttöliittymässä placeholder-attribuutilla määriteltyjen opasteiden avulla (kuva 59).



Kuva 59. Käyttäjän ohjeistaminen

Tarkastelussa havaittiin, että käyttöliittymä täyttää valtaosan Nielsenin heuristiikkalistan kohdista. Muutama käytettävyysongelma löydettiin, mutta pääasiassa ne ovat vakavuusasteeltaan lieviä tai kosmeettisia. Validoinnin puutetta listaus- ja muokkausnäkymissä voidaan pitää vakavana käytettävyysongelmana, joka pitää korjata jatkokehityksessä. Ongelmista huolimatta voidaan kuitenkin todeta käytettävyyden olevan kohtuullisen hyvällä tasolla.

5 Pohdinta

Aiheena pilvipohjaisen koodileikkeiden hallintasovelluksen rakentaminen oli todella mielenkiintoinen ja se tarjosi hyvät edellytykset ammatilliselle kasvulle verkkosovelluskehityksen alalla. Tiesin heti alusta alkaen joutuvani tutkimaan ja perehtymään useisiin tekniikoihin ja työkaluihin, joista en nimeä lukuun ottamatta tiennyt käytännössä mitään. Projektin aikana modernit verkkosovelluskehityksen tekniikat, työkalut sekä menetelmät tulivatkin hyvin tutuiksi.

Jälkeenpäin ajateltuna aihe oli liian laaja ja teknisesti haastava kokonaisuus alkutasooni nähden. Laajuus ja tekninen haastavuus tekivät raportin rajaamisesta ja kirjoittamisesta vaikeaa. Myös sovelluksen kehittämiseen kului enemmän aikaa kuin alussa kuvittelin, mutta raportin kirjoittaminen oli kuitenkin pääsyy sille, että aikataulua jouduttiin projektin aikana venyttämään useampaan otteeseen. Alkuperäisestä aikataulutavoitteesta jäätiinkin lopulta huomattavasti jälkeen.

Teknisissä valinnoissa onnistuttiin huomattavasti aihevalintaa paremmin. Valitut tekniikat tukivat sovelluksen kehitystyötä erittäin hyvin. Kun tarvittava taitotaso työkalujen ja tekniikoiden käyttämisessä oli saavutettu, sovelluksen kehitys oli nopeaa ja työskentely mielekästä. Toisaalta tekniikoiden opetteluun kului paljon aikaa; ongelmana oli valittujen tekniikoiden jyrkkä oppimiskäyrä ja tehokkaan kehitysympäristön muodostamiseen tarvittavien työkalujen määrä. Ajattelin kuitenkin näiden tekniikoiden ja työkalujen opettelua investointina sovelluksen jatkokehityksen ja oman työskentelyni tehostamiseen tulevaisuudessa.

Aloittaessani työskentelyn tämän projektin parissa taitoni front-end-kehityksen puolella rajoittuivat hyvin alkeellisten tekniikoiden käyttämiseen. Tämän projektin parissa työskentely on vienyt taitotasoani verkkosovelluskehityksen alalla todella paljon eteenpäin. Varsinkin front-end-kehityksen puolella kehitystä on tapahtunut valtavasti. Suurimmat oppikokemukset ja ahaa-elämykset ovatkin liittyneet ehdottomasti JavaScriptiin ja käyttöliittymä- sekä käyttäjäkokemussuunnitteluun. Reactin, Reduxin, Webpackin ja muiden tekniikoiden sekä työkalujen kanssa työskentely avasi silmäni sille, miten tehokasta ja miellyttävää front-end-kehityksestä voidaan saada modernin teknologiapinon avulla. Vaikka React ja Redux eivät pysyisi relevantteina tekniikoina tulevaisuudessa, uskon että pystyn hyödyntämään projektin aikana oppimiani menetelmiä jossain muodossa myös tulevaisuudessa.

Sovelluksessa on paljon jatkokehityskohteita. Jatkokehityksessä keskitytään pääasiassa jo olemassa olevien toiminnallisuuksien, käyttöliittymän sekä tietoturvan kehittämiseen. Edellä mainituista kehittämiskohteista varsinkin tietoturvan kehittäminen on ensisijaisen tärkeää, mikäli sovellus halutaan tulevaisuudessa julkaista yleiseen käyttöön.

Jo olemassa olevan sisällön lisäksi kehitetään muutama uusi toiminnallisuus. Oleellisimmat toiminnallisuudet, joita ei vielä tähän sovelluksen versioon toteutettu, ovat gistien kommentointi ja versionhallinta jossain muodossa. Rajoittava tekijä uusien toiminnallisuuksien lisäämisessä on Gist API:n tarjoamien toiminnallisuuksien rajallisuus. Kommentointia ja versionhallintaa

lukuun ottamatta Gist API:n tarjoamat toiminnallisuudet on jo hyödynnetty nykyisessä versiossa. Uusien toiminnallisuuksien lisääminen edellyttäisi huomattavia lisäyksiä nykyiseen ympäristöön, esim. oman tietokannan lisäämistä.

Lähteet

Aalto-yliopisto. 2017. Heuristisen arvioinnin muistilista. http://www.uiah.fi/mediastudio/survey4/liitea1.html. 7.2.2017.

Ace. 2016. About ACE. https://ace.c9.io/#nav=about. 20.9.2016.

Dabit, N. 2016. Beginner's Guide to React Router. 5.4.2016. https://medium.com/@dabit3/beginner-s-guide-to-react-router-53094349669#.mf8txcpvh. 25.10.2016.

Facebook. 2016a. React. https://facebook.github.io/react. 19.11.2016.

Facebook. 2016b. Thinking in React. https://facebook.github.io/react/docs/thinking-in-react.html. 19.11.2016.

Facebook. 2016c. Tutorial: Intro To React. https://facebook.github.io/react/tutorial/tutorial.html. 19.11.2016.

Git. 2017. https://git-scm.com/book/fi/v1/Alkusanat-Versionhallinnasta. 18.2.2017.

GitHub. 2016a. About gists. https://help.github.com/articles/about-gists. 19.11.2016.

GitHub. 2016b. Gist API. https://developer.github.com/v3/gists. 19.11.2016.

GitHub. 2016c. Oauth. https://developer.github.com/v3/oauth. 19.11.2016.

Krajka, B. 2016. The Difference Between Virtual DOM and DOM. 12.10.2016. http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/. 19.11.2016.

Kubacak, C. 2016. Getting Started With Redux. https://scotch.io/bar-talk/getting-started-with-redux-an-intro. 6.11.2016.

Mozilla. 2017. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. 18.2.2017

Mozilla. 2016. Using Fetch. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. 13.11.2016.

Nielsen, J. 1995. 10 Usability Heuristics for User Interface Design. https://www.nngroup.com/articles/ten-usability-heuristics/. 7.2.2017.

React Router. 2016. React Router. https://github.com/ReactTraining/react-router. 25.10.2016.

Redux. 2016a. Motivation. http://redux.js.org/docs/introduction/Motivation.html. 10.9.2016.

Redux. 2016b. Basic Reducer Structure. http://redux.js.org/docs/recipes/reducers/BasicReducerStructure.html. 6.11.2016.

Redux. 2016c. Actions. http://redux.js.org/docs/basics/Actions.html. 15.9.2016.

Redux. 2016d. Three Principles. http://redux.js.org/docs/introduction/ThreePrinciples.html. 10.9.2016.

Redux. 2016e. Reducers. http://redux.js.org/docs/basics/Reducers.html. 15.9.2016.

Redux. 2016f. Store. http://redux.js.org/docs/basics/Store.html. 15.9.2016.

Redux. 2016g. Data Flow. http://redux.js.org/docs/basics/DataFlow.html. 11.9.2016.

Redux. 2016h. Async Flow. http://redux.js.org/docs/advanced/AsyncFlow.html. 14.11.2016.

Redux. 2016i. Async Actions. http://redux.js.org/docs/advanced/AsyncActions.html. 14.11.2016.

Redux. 2016j. Organizing State. http://redux.js.org/docs/faq/OrganizingState.html. 14.11.2016.

Redux. 2016k. Usage With React. http://redux.js.org/docs/basics/UsageWithReact.html. 8.9.2016.

Stack Overflow. 2016. How to dispatch a redux action with a timeout? http://stackoverflow.com/questions/35411423/how-to-dispatch-a-redux-action-with-a-timeout/35415559#35415559. 26.9.2016.

Webpack. 2016a. What is Webpack? http://webpack.github.io/docs/what-is-webpack.html. 19.11.2016.

Webpack. 2016b. Using Loaders. http://webpack.github.io/docs/using-loaders.html. 19.11.2016.

Webpack. 2016c. Webpack Dev Server. https://webpack.github.io/docs/webpack-dev-server.html. 19.11.2016.

Webpack. 2016d. Build Performance. http://webpack.github.io/docs/build-performance.html. 19.11.2016.

Wikipedia. 2017a. Create, read, update and delete. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. 18.2.2017.

Wikipedia. 2017b. ECMAScript. https://en.wikipedia.org/wiki/ECMAScript. 18.2.2017.

Wikipedia. 2017c. JSON. https://en.wikipedia.org/wiki/JSON. 18.2.2017.

Wikipedia. 2016a. GitHub. https://fi.wikipedia.org/wiki/GitHub. 18.2.2017.

Wikipedia. 2016b. REST. https://fi.wikipedia.org/wiki/REST. 18.2.2017.

Wikipedia. 2016c. Single-page application. https://en.wikipedia.org/wiki/Single-page_application. 25.10.2016.

Wikipedia. 2016d. ACE (editor). https://en.wikipedia.org/wiki/Ace_(editor). 20.9.2016.

Wikipedia. 2015. Ohjelmointirajapinta. https://fi.wikipedia.org/wiki/Ohjelmointirajapinta. 18.2.2017.

W3C. 2005. Document Object Model (DOM). https://www.w3.org/DOM/. 18.2.2017.