WEB-SOVELLUS NODE.JS-YMPÄRISTÖSSÄ

Esimerkkinä chat-sovellus

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutus

Kevät 2022

Paavo Kalliala

Tietojenkäsittelyn koulutus | Tiivistelmä
Tekijä Paavo Kalliala | Vuosi 2022
Työn nimi Web-sovellus Node.js-ympäristössä
Ohjaaja Lasse Seppänen

Opinnäytetyön päätavoitteena oli kehittää tekijän osaamista ja oppia uutta tekniikkaa. Työ käsittelee yleisesti web-sovelluksia ja sovelluksen kehittämiseen käytettäviä työkaluja. Aiheena oli tutustua web-sovelluksen kehittämiseen JavaScript-ohjelmointikielellä palvelinpuolen Node.js-ympäristössä. Idea aiheeseen tuli tekijän kiinnostuksesta kehittämiseen ja Node.js:n suosioon web-kehityksessä. Opinnäytetyön aihe on tekijän itse keksimä.

Teoriaosuudessa tutustutaan web-sovelluksen toimintaperiaatteisiin, sovelluksen osa-alueisiin sekä tekniikoihin. Työn pääaiheena oleva palvelinpuolen Node.js-ympäristö on teoreettisen osuuden laajin kokonaisuus. Opinnäytetyö on toiminallinen ja työssä toteutetaan esimerkkinä yksinkertainen projekti, jonka tuloksena syntyi chat-sovellus. Aineisto koostui päiväkirjasta, johon työn aikana kirjattiin projektin vaiheita. Aineistoa analysoitiin työn tuloksissa.

Työn aikana huomattiin kuinka Node.js-alustalla kehittäminen on tehty helpoksi ja yksinkertaisen web-sovelluksen kirjoittamiseen tarvitaan vain muutama rivi koodia. Työn toiminnallisen osuuden tuloksena syntyi yksinkertainen chat-sovellus, joka hyödynsi Node.js:n moduuleita Express ja socketIO. Chat-sovelluksen kehittämistä jatketaan opinnäytetyöprosessin jälkeen. Tekijä on tyytyväinen työn tulokseen siihen nähden, että aikataulu koko opinnäytetyön prosessiin oli tiukka. Työn pohjalta kiinnostus web-kehittämiseen kasvoi ja tekijä aikoo toteuttaa Node.js projekteja tulevaisuudessa.

Avainsanat Node.js, JavaScript, Sovellus, Chat, Web-kehitys

Sivut 30 sivua ja liitteitä 9 sivua

Degree Programme in Business Information Technology
Author Paavo Kalliala
Subject Web application using Node.js environment
Supervisors Lasse Seppänen

Abstract
Year 2022

The main purpose of this thesis was to improve the author’s skills and learn new techniques. The thesis focuses on web applications and the tools used to develop them. The main topic was to explore the development of the web application using JavaScript in a server-side Node.js environment. The idea for the topic came from the author’s interest in development and because of the popularity of the Node.js in web-development. The topic of the thesis was invented by the author himself.

The theoretical part introduces the principles of a web application, components, and techniques. The server-side Node.js environment, which is the main topic of the thesis, is the most extensive part of the theoretical part. The thesis is action-oriented and an example, a simple project was implemented, resulting in a chat application. The material collected during the work was in the form of a diary. The diary was used to record the steps of the project. The material was analyzed in the result of the project.

During the process, we noticed how the Node.js platform made it easy to develop and it takes only few lines of code to write a simple web application. The functional part of the thesis resulted in a simple chat application that utilized the Node.js modules Express and SocketIO. The author is satisfied with the result of the work considering the tight schedule for the whole thesis process. Based on the thesis, the interest in web development increased and the author plans to implement Node.js projects in the future.

Keywords Node.js, JavaScript, Application, Chat, Web development

Pages 30 pages and appendices 9 pages

Sanasto

API	Ohjelmointirajapinta, jonka avulla sovellukset pystyvät kommunikoimaan toistensa kanssa
Backend	Sovelluksen palvelinpuoli. Käyttäjälle näkymättömissä. Hoitaa sovelluksen logiikan ja tiedon varastoinnin
CSS	Cascading Style Sheets. Verkkosivun ulkoasun tyylittely
Frontend	Selaimessa toimiva sovelluksen osuus (HTML, CSS, JavaScript)
HTML	HyperText Markup Language, verkkosivujen määrittelykieli
HTTP	Hypertext Transfer Protocol, hypertekstin siirtoprotokolla, jota selaimet ja www-palvelimet käyttävät tiedonsiirtoon.
JSON	JavaScript Object Notation, tiedostomuoto tiedon välitykseen.
Kirjasto	Sisältävät valmiiksi kirjoitettua koodia. Helpottaa ja nopeuttaa sovelluksen kirjoittamista.
Node	Node.js, Palvelinpuolen JavaScript ajoympäristö
NPM	Node Package Manager. Pakettien hallinta järjestelmä, jolla hallitaan Noden työkalujen asennuksia
Selain	Sovellus, joka tarjoaa tavan katsella ja olla vuorovaikutuksessa Internetissä olevan informaation kanssa.
TCP	Transmission Control Protocol. TCP-yhteyksien avulla tietokoneet voivat lähettää toisilleen tietoa luotettavasti
Tietokanta	Tiedon tallentamiseen, lukemiseen ja hallinnointiin.

Sisällys

1 Johdanto 1
2 Web-sovellus 2
2.1 Selainpuoli 2
2.2 Palvelinpuoli 4
2.3 HTML 5
2.4 CSS 6
2.5 JavaScript 6
2.5.1 Historia 6
2.5.2 ECMAScript 7
2.5.3 Ohjelmointikieli 7
2.6 Tietoturva 8
3 Node.js 10
3.1 Historia 10
3.2 Käyttökohteet 10
3.3 Non-blocking 11
3.4 Asynkronisuus 11
3.5 Moduulit 12
4 Palvelinpuolen työkalut 13
4.1 NPM 13
4.2 Express.js 14
4.3 SocketIO 14
4.4 Nodemon 15
4.5 Visual Studio Code 16
4.6 Versionhallinta GitHubilla 16
5 Chat-sovelluksen toteutus 17
5.1 Asennukset 17
5.2 Toiminallisuus 17
5.3 Sovelluksen rakenne 20
5.4 Ohjelmakoodi 20
5.4.1 Projektin aloitus 21
5.4.2 Express web-palvelin 21

5.4.3 Socket.IO ... 22
5.4.4 Käyttöliittymä ... 22
5.4.5 Kirjautuminen ... 22
5.4.6 Socket-yhteys ... 23
5.4.7 Session ... 23
5.4.8 Käyttäjät ... 24
5.4.9 Viesti ... 24
5.4.10 Privaattiviestit ... 24
5.4.11 Palvelimen käynnistys ... 25
6 Tulokset ... 26
7 Johtopäätökset ja pohdinta ... 27
8 Yhteenveto ... 28
Lähteet ... 29

Kuvat, komennot, listat ja ohjelmakoodit

Komento 1. Moduulin asennus NPM ... 13

Kuva 1. Staattinen verkkosivu ... 3
Kuva 2. Dynaaminen verkkosivu ... 3
Kuva 3. Noden tapahtumasilmukka. ... 12
Kuva 4. WebSocket-yhteys ... 15
Kuva 5. Käyttäjänimen syöttäminen ... 18
Kuva 6. Chat-ikkuna ... 18
Kuva 7. Privaatti-ikkuna. ... 19
Kuva 8. Chat-sovelluksen toimintakaavio. ... 19
Kuva 9. Sovelluksen rakenne. ... 20

Lista 1. Top-10 listaus web-sovellusten haavoittuvuuksista. ... 9

Ohjelmakoodi 1. Esimerkki HTML-koodista ... 5
Ohjelmakoodi 2. Esimerkki CSS-koodista ... 6
Ohjelmakoodi 3. Esimerkki JavaScript-kielestä ... 7

Ohjelmakoodi 4. Express-moduulin käyttöönotto 12
Ohjelmakoodi 5. Yksinkertainen HTTP-palvelin Express moduulilla 14
Ohjelmakoodi 6. NPM skriptit 16
Ohjelmakoodi 7. HTTP-palvelin Expressillä 21
Ohjelmakoodi 8. SocketIO:n käyttöönotto 22
Ohjelmakoodi 9. Express static.. 22
Ohjelmakoodi 10. Socket-yhteys 23
Ohjelmakoodi 11. Session tallennus ja lähetys 23
Ohjelmakoodi 12. Käyttäjien haku.. 24
Ohjelmakoodi 13. Viestin lähettäminen. 24
Ohjelmakoodi 14. NPM-skriptit. 25

Liitteet

Liite 1 Aineistonhallintasuunnitelma
Liite 2 Ohjelmakoodi: package.json
Liite 3 Palvelinpuolen ohjelmakoodi: index.js
Liite 4 Selainpuolen ohjelmakoodi: index.html

1 Johdanto

JavaScript-kielen suorittaminen pelkästään selaimessa on historiaa. Nyt se on valloittamassa suurta osaa sovelluskehityksestä. JavaScriptin suuresta suosiosta johtuen päätin itsekin lähteä tutustumaan JavaScript-kielellä kehittämiseen.

Opinnäytetyössä tutustutaan web-sovellukseen, sekä syvällisemmin palvelinpuolen työkaluihin. Toteutusvaiheessa tehdään pienimuotoinen projekti, jossa toteutetaan palvelinpuolen web-sovellus JavaScript-kiellellä.

Toteutuksessa käytetään JavaScriptin suorittamiseen palvelinpuolella yleisesti käytettyä Node.js-ajoympäristöä. Nodella pystytään suorittamaan JavaScript-kieltä palvelinpuolella. Noden ja sen työkalujen avulla on helppo toteuttaa kevyt web-palvelinsovellus. Toteutuksessa keskitytään palvelinpuoleen, mutta tehdään myös yksinkertainen käyttöliittymä selainpuolelle sovelluksen toiminnallisuuksien esittelyä varten.

Työn teen itselleni ja sen päätavoitteena on kehittää omaa osaamistani aloittelevana kehittäjänä, opettelemalla perusteet JavaScript-kielestä sekä palvelinpuolen JavaScript-ajoympäristöstä Nodesta. Lisäksi tavoitteena on tutkimuskysymyksiin vastaaminen ymmärrettävästi ja selkeässä muodossa.

Tutkimuskysymykset keskittyvät web-sovellukseen ja Node-ympäristöön:

- Mikä on web-sovellus?
- Mitä eroa on perinteisellä nettisivulla ja web-sovelluksella?
- Minkälaisia tietoturvauhkia web-sovellukseen kohdistuu?
- Mikä on Node.js?
- Mitä Noden kirjastoja käytetään chat-sovelluksen kehittämiseen?

2 Web-sovellus

Web-sovellus on sovellus, jota pääsee käyttämään erilaisilla nettiselaimilla laitteesta riippumatta. Peruskäyttäjälle Web-sovellus voi näyttää samalta kuin vanhat perinteiset staattiset web-sivut. Web-sovellus koostuu selainpuolesta ja palvelinpuolesta. Selaimen näkymä toimii käyttöliittymänä. Palvelin hoitaa taustalla sovelluksen logiikan ja tiedon tarjoamisen käyttöliittymälle. Verkkosivua, joka suorittaa ohjelmakoodia palvelimella tai selaimessa, sanotaan usein web-sovellukseksi. (Martin, 2021)

Web-sovellukset ovat koko ajan suositumpia. Verrattuna työpöytäsovelluksiin, selaimessa toimivat sovellukset ovat vähemmän riippuvaisia laitteistosta. Selaimella toimivia sovelluksia ei myöskään tarvitse asentaa, sekä niiden päivittäminen on helpompaa kuin työpöytäsovellusten. (Martin, 2021)

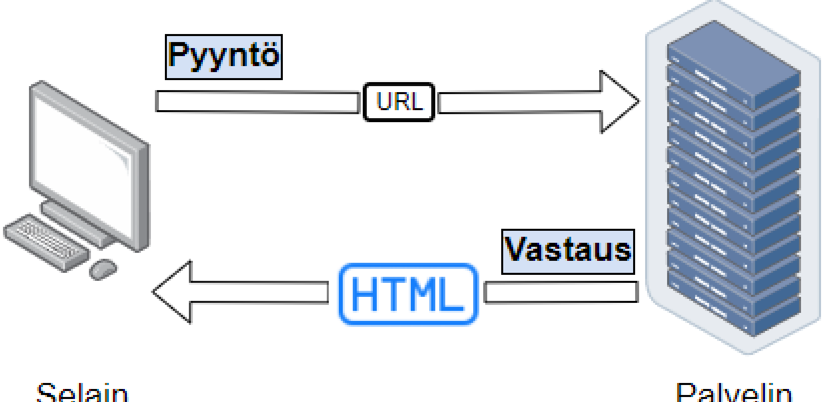
Web-sovellusten huonoja puolia ovat, että ne tarvitsevat usein internet-yhteyden. Tietoturva on myös potentiaalinen riski, kun liikutetaan tietoja internetin ylitse. Pilvipalveluita käyttäessä ei välttämättä itse pääse vaikuttamaan miten tietoja varastoidaan.

2.1 Selainpuoli

Sovelluksen käyttöliittymä eli selainpuoli on usein kehitetty käyttämällä yleisimpiä kieliä, HTML, CSS ja JavaScript, joita eniten käytetyimmät selaimet tukevat. JavaScriptin osuus kaikista web-sovelluksista on 97,9 % (w3techs, n.d.). Peruskäyttäjälle käyttöliittymä voi näyttää perinteiseltä verkkosivulta. Taustalla tapahtuu kuitenkin käyttäjälle näkymätöntä toiminnallisuutta. (Karhu Helsinki Oy, n.d.)

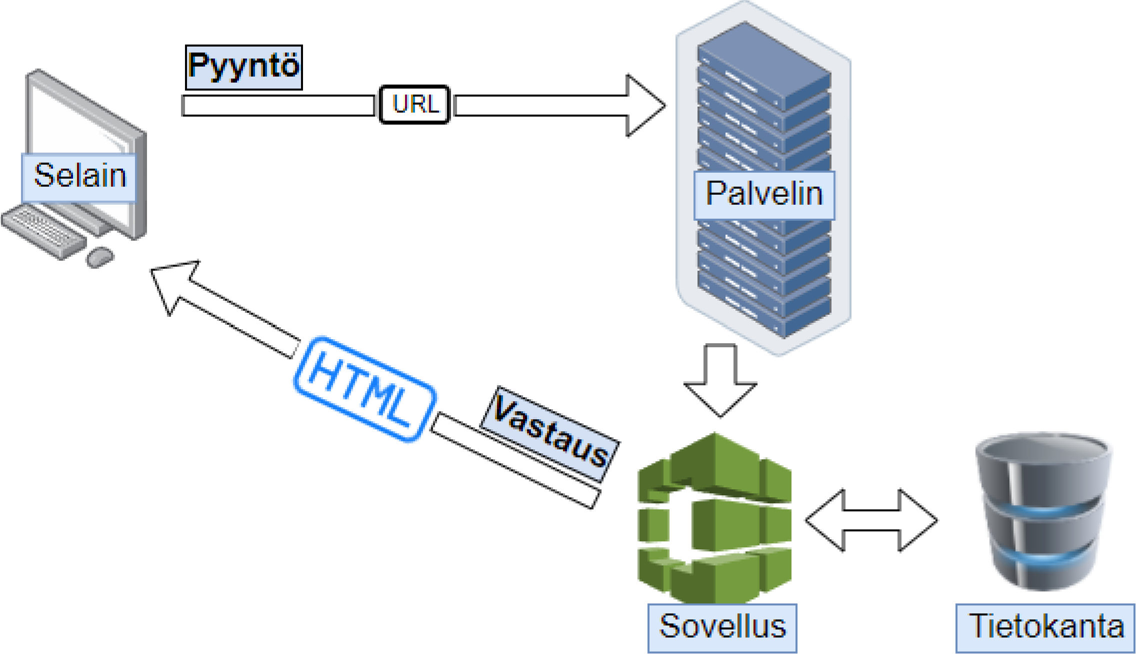
Perinteisellä **staattisella** verkkosivulla tarkoitetaan dokumenttia, joka on julkaistu internetissä. Dokumentit ovat saatavilla palvelimella, josta selaimen käyttäjät voivat hakea niitä pyynnöstä. Esimerkissä (Kuva 1) käyttäjä tekee pyynnön palvelimelle menemällä verkkosivulle ja palvelin lähettää vastauksena HTML-dokumentin takaisin. Verkkosivut näyttävät samalta kaikille käyttäjille ja sivuston sisältö ei vaihdu, jos sitä ei erikseen muokata palvelimella. Staattisiin sivuihin tarvitaan vain HTML-merkintäkieltä. (Karhu Helsinki Oy, n.d.)

Kuva 1. Staattinen verkkosivu



Web-sovellukset ovat **dynaamisia** verkkosivuja. Dynaamiset verkkosivut (Kuva 2) eivät ole valmiita dokumentteja palvelimella, vaan ne luodaan samalla kun selain tekee pyynnön palvelimelle. Dynaaminen verkkosivu voi sisältää myös staattisia sivuja ja vain sivujen sisältöä päivitetään. Käyttäjä tekee pyynnön selaimella palvelimelle. Palvelin tunnistaa pyynnön ja alkaa suorittamaan pyyntöä vastaavaa ohjelmaa, joka esimerkiksi tarvittaessa hakee tietokannasta tietoa tai suorittaa jonkun laskun. Suoritettuaan ohjelman, palvelin lähettää selaimelle vastauksen. (Peltomäki, 2020)

Kuva 2. Dynaaminen verkkosivu



Web-sovelluksen selaimen näkymä eli käyttöliittymän sisältö, voidaan koostaa selaimessa suoritettavan JavaScript-koodin avulla. Palvelin voi myös lähettää vain JSON-muotoista tietoa ja tämän selain koostaa käyttäjälle suotuisaksi verkkosivuksi. Käyttöliittymä voidaan renderöidä myös palvelinpäässä, jolloin palvelin lähettää valmiin HTLM-tiedoston selaimella. (Peltomäki, 2020)

Viime vuosina suosituksi on tullut yhden sivun ratkaisut eli **SPA** (Single Page Application). Käyttäjä pysyy yhdellä sivulla, jota muokataan käyttäjän toimien mukaisesti. Palvelin vastaa pyyntöön yhdellä HTML-sivulla. Sivua muokataan selaimessa toimivalla ohjelmakoodilla, jotka palvelin lähettää selaimella suoritettavan ohjelmakoodin pyynnöstä. Koko sivu ei siis päivity, vain sen sisältö. (Peltomäki, 2020)

2.2 Palvelinpuoli

Web-sovelluksen toiminallisuuden hoitaa palvelinpuoli eli tietokone, johon on asennettu Web-palvelimen tarjoama ohjelmisto. Ensin palvelin tarjoaa selaimelle sovelluksen käyttöliittymän, käyttäjän syötettyä palvelimen osoite selaimen osoiteriville. Käyttöliittymä lähettää palvelimelle pyyntöjä, kun käyttäjä tekee toimintoja käyttöliittymässä. Palvelimen tehtävä on vastata selaimelta tuleviin pyyntöihin. Pyynnöt voivat olla esimerkiksi tietokannan käsittelyä tai laskutoimituksen suorittamista. Palvelin suorittaa pyynnön ja lähettää vastuksen selaimelle. Selainpuolen koodit määrittävät, mitä vastaanotetulle informaatiolle tehdään. Vastaus on yleensä HTML-sivu tai tiedosto. Nykypäivän sovellukset siirtävät tietoa JSON-muodossa. Selaimen ja palvelimen välisiin tiedonsiirtoihin käytetään HTTP-protokollaa. (Peltomäki, 2020)

Web-kehittäjä, joka hallitsee frontendin eli palvelinpuolen, sekä backendin eli selainpuolen, sanotaan Full Stack -kehittäjäksi. Full Stack -kehittäjä pystyy toteuttamaan tietokannan, sovelluksen logiikan, sekä sen ulkoasun eli koko sovelluksen. Stack viittaa järjestelmän rakenteen jaotteluun eli mitä teknisiä ratkaisuja on käytetty sovelluksen toteuttamiseen. Esimerkkinä MEAN-stack, jonka lyhenne tulee tekniikoista MongoDB, Express, Angular sekä Node. MEAN-stackissa kaikki on ohjelmoitu JavaScriptillä, mikä helpottaa kehittämistä. MongoDB on tietokanta. Angular

on selainpuolen kehys. Node on palvelinpuolen alusta ja Express sen kirjasto, joka toteuttaa HTTP-palvelimen. (Peltomäki, 2020)

Palvelinpuolet ovat kompleksisempia ja lähestymistapoja on monenlaisia. Palvelinpuolen kokonaisuuden voi kehittää käyttäen erilaisia ympäristöjä, ohjelmointikieliä ja tietokantoja. Tässä työssä kuitenkin keskitytään JavaScript-kielellä ohjelmoitavaan Node.js-ympäristöön.

2.3 HTML

HTML on merkintäkieli, jolla useimmat verkkosivut on kirjoitettu. HTML:n avulla esitetään verkkosivujen rakenne ja sisältö. Selaimen moottori lukee HTML-koodia ja tulkitsee koodia selaimessa käyttäjälle ymmärrettävässä muodossa. Selain valmistajia on useita ja jokaisella on oma HTML-moottorinsa. Eri valmistajien selaimet voivat hieman poiketa toisen tavasta esittää tulkattua koodia. (W3schools, 2022b)

HTML-dokumentti rakennetaan HTML-elementeillä (Ohjelmakoodi 1). HTML-dokumentti alkaa aina tagilla <html> ja loppuu lopetustagilla </html>. Elementit kirjoitetaan aloitus- ja lopetustagilla. Kaikilla elementeillä ei ole lopetustagia. Tagien väliin kirjoitetaan elementin sisältö. Elementissä voi olla sisäisiä elementtejä, jotka koodia kirjoittaessa usein sisennetään. Sisennykset auttavat kehittäjää löytämään mistä elementti alkaa ja loppuu. (W3schools, 2022b)

Ohjelmakoodi 1. Esimerkki HTML-koodista

```
<html>
<body>
    <h1> OTSIKKO </h1>
    <p> Tähän voi kirjoittaa tekstiä </p>
</body>
</html>
```

2.4 CSS

HTML-elementtien muotoilu ja sijainnit selaimessa kirjoitetaan yksinkertaisella CSS-kielellä. Yleensä CSS-tyylittelyt kirjoitetaan omaan tiedostoonsa. CSS-tyylittelytiedosto määritellään HTML-dokumentin alkuun, head-tagin sisään. CSS-kielen syntaksi koostuu elementin valitsimesta ja aaltosulkeiden sisään syötettävistä pareista, joissa määritellään elementin ominaisuus ja sen arvo. Ohjelmakoodi 2 esimerkissä määritetään otsikko eli h1-tagin väri punaiseksi. (W3schools, 2022a)

Ohjelmakoodi 2. Esimerkki CSS-koodista

```
h1 {
    color: red;
}
```

2.5 JavaScript

JavaScript on monikäyttöinen ohjelmointikieli, joka on alkujaan suunniteltu tekemään verkkosivuista interaktiivisia. JavaScript mahdollistaa esimerkiksi verkkosivun sisällön lataamisen ja päivittämisen ilman, että käyttäjän tarvitsee kirjoittaa osoitepalkkiin tai tehdä erillisiä pyyntöjä palvelimelle.

2.5.1 Historia

JavaScript on kehitetty alkujaan Netscape-selaimeen 1990-luvulla. Alun perin sen tarkoitus oli lisätä dynaamisuutta verkkosivuille. JavaScript-kielen syntaksilla ei ole mitään tekemistä Java-kielen kanssa, vaikka nimessä samanlaisuutta. Nimi otettiin markkinointi syistä, koska Java oli siihen aikaan suuressa suosiossa. Samaan aikaan Microsoftilla oli omansa nimeltään JScript, joka oli lähes samanlainen kuin JavaScript. Kielien kehittyessä Netscape ja Microsoft taistelivat yhteensopivuus ongelmista ja tähän yhteensopimattomuus ongelmaan haluttiin ratkaisu. Netscape siirsi JavaScriptin hallinnan Euroopan tietokonevalmistajien yhdistykselle ECMA:lle. Yhdistys muodosti standardin ECMAScript. Nykyään JavaScriptin tavaramerkin omistaa Oracle. (Wirfs-Brock & Eich, 2020)

2.5.2 ECMAScript

ECMA eli European Computer Manufacturers Association on eurooppalainen standardointijärjestö, jolla on monia tietotekniikka-alan kansainvälisi standardeja. Yksi merkittävämpiä standardeja on ECMAScript. ECMAScript-standardin tarkoitus on varmistaa JavaScriptin toimivuus eri selaimissa. ECMAScriptin kehitys on avointa ja kehitykseen pääsee osallistumaan GitHubissa. (Mozilla, 2022)

2.5.3 Ohjelmointikieli

Oliopohjainen JavaScript on ohjelmointikielenä monimuotoinen ja sillä kehittäminen on nopeampaa kuin esimerkiksi Java-kielellä. Koodissa käytetään olioita eli objekteja. Olioilla voi olla ominaisuuksia ja funktioita. JavaScript-kielessä ei käytetä tyyppijärjestelmää vaan tyyppi päätellään suorituksen aikana. Ohjelmakoodin virheiden käsittely on anteeksiantava ja tämä tuokin välillä haasteita koodivirheitä etsiessä. (Wirfs-Brock & Eich, 2020)

Ohjelmakoodissa 3 on esimerkkinä JavaScript-kielellä luotu olio. Oliolla **henkilo** on ominaisuutena **nimi**, **paino**, **pituus**, sekä funktiona **tervehtii**. Olion ominaisuuksia voidaan kutsua esimerkiksi henkilo.paino, joka palauttaa arvon 34. Funktioon pääsee käsiksi henkilo.tervehtii(), joka palauttaa ”Moi! Olen Jaakko”.

Ohjelmakoodi 3. Esimerkki JavaScript-kielestä

```
var henkilo = {
  nimi: ["Jaakko", "Parantainen"],
  paino: 34,
  pituus: 188,
  tervehtii: function() {
    return "Moi! Olen " + this.nimi[0] + ".";
  },
}
```

2.6 Tietoturva

Web-sovelluksien yleistyessä myös niiden tietoturvauhat ovat kehittyneet. Toisin kuin työpöytäsovelluksiin, netissä oleviin web-sovelluksiin pääsee melkein kuka vaan kokeilemaan tietoturva-aukkoja. Web-sovellusten haavoittuvuuksia on erilaisia ja ne voidaan jakaa neljään pääluokkaan: syötteenkäsittelyn liittyvät heikkoudet, käyttäjien tunnistamiseen ja valtuuttamiseen liittyvät haavoittuvuudet, tiedon suojaaminen ja sovelluksen käyttämiin komponentteihin ja alustaan liittyvät heikkoudet. (Second Nature Security, 2018)

Web-sovelluksia on syytä kehittää siten että, yleisimpiä haavoittuvuuksia huomioidaan ja testataan. Syötteet validoidaan, että niihin ei pääse tekemään lisäyksiä. Käyttäjien tekemät pyynnöt on hyvä auktorisoida ja tunnistaan. Sovelluksessa oleva tai keräämä tieto on syytä analysoida ja suunnitella siten tiedon suojaaminen. Käyttäjien tunnistaminen sovellukseen nähden riittävän vahvalla tunnistamisella. (Second Nature Security, 2018)

The Open Web Application Security Project (OWASP) pitää listaa (Lista 1) yleisimmistä tietoturvauhista. OWASP on kansainvälinen yhteisö, jonka tarkoituksena on tiedottaa tieturvariskeistä ja siten edistää turvallisten web-sovelluksien kehitystä. Suosituimpia haavoittuvuuksia ovat muun muassa pääsynhallinnan valvonta (Broken Access Control), jolla valvotaan, että käyttäjät eivät pääse suorittamaan oikeuksiensa ulkopuolisia toimintoja. Myös injektiot ovat yleisiä. Injektiossa esimerkiksi SQL-syötteeseen lisätään koodia, jolla kalastellaan tietoa tietokannasta. (OWASP, n.d.)

Lista 1. Top-10 listaus web-sovellusten haavoittuvuuksista.

2021

A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
A10:2021-Server-Side Request Forgery (SSRF)*

* From the Survey

3 Node.js

Node.js eli Node mahdollisti JavaScriptin käytön palvelinpuolelle. Googlen Chrome-selaimen moottoria käyttävällä Nodella on helppo luoda web-palvelin vain muutamalla rivillä JavaScript-koodia (Ohjelmakoodi 5). Paljon rinnakkaisuutta vaativiin verkkosovelluksiin Node on parhaimmillaan. Noden yksisäikeisyyden vuoksi, sitä ei suositella verkkosovelluksiin, jotka vaativat raskasta prosessointia. (Peltomäki, 2020)

3.1 Historia

Node kehitettiin avoimen lähdekoodin JavaScript-alustaksi, palvelinsovellusten rakentamiseen, jotka pystyisivät käsittelemään suuren määrän samanaikaisia yhteyksiä. Noden toteutus koostui Googlen V8 JavaScript -moottorista, CommonJS Module -latausohjelmasta ja joukosta C-kielellä toteutettuja moduuleja. (Wirfs-Brock & Eich, 2020)

Vuonna 2009 toukokuussa julkaistiin ensimmäinen julkinen versio. Isommin huomiota Node sai vasta, kun sen kehittäjä Ryan Dahl esitteli sen jsconf.eu-konferenssissa. Melko pian tämän jälkeen Ryan Dahl palkattiin Joyentiin, joka hallinnoi ja tuki Noden jatkokehitystä, kunnes vastuu siirrettiin Node Foundation -säätiölle vuonna 2015. (Wirfs-Brock & Eich, 2020)

Node suunniteltiin palvelinsovellusten rakentamiseen, mutta siitä tulikin alusta, joka mahdollisti JavaScriptin käytön yleiskäyttöisenä ohjelmointikielenä monilla eri alustoilla, myös sulautetuissa järjestelmissä. Node mahdollisti sen, että web-ohjelmoivat pystyivät siirtämään taitonsa selainympäristöstä erilaisiin sovelluksiin. Alun perin web-kehittäjät käyttivät JavaScript-kieltä, koska muuta vaihtoehtoa ei ollut. Nykyään monet kehittäjät haluavat käyttää mieluimmin JavaScriptiä. (Wirfs-Brock & Eich, 2020)

3.2 Käyttökohteet

Noden tärkeimpiä käyttökohteita ovat Streaming-tietovirtaa hyödyntävät sovellukset, tapahtumapohjaiset sovellukset, yhden sivun sovellukset (Single-Page Application), sekä

reaaliaikaiset chat-sovellukset. Node on parhaimmillaan reaaliaikaisissa verkkosovelluksissa, joissa käytetään kaksisuuntaisia yhteyksiä käyttäjän ja palvelimen välillä. Node käsittelee I/O-tehtäviä nopeasti käyttäen mm. sisäänrakennettuja WebSocket-API- sekä Event API-rajapintoja. Nopeuden ja tehokkuuden ansiosta Nodesta onkin tullut sovelluskehittäjien suosikkivalinta reaaliaikaisten ja tapahtumapohjaisten sovellusten rakentamiseen. (Peltomäki, 2020)

3.3 Non-blocking

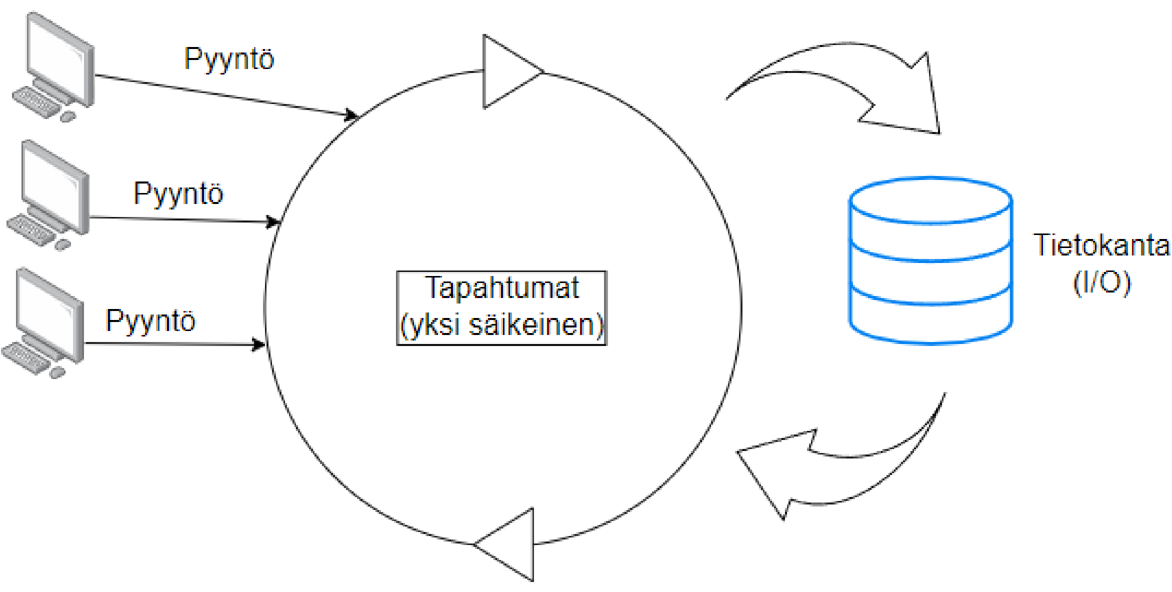
Kyselyt tietokantaan ja tiedon välitys selaimelle eli käyttöliittymälle onnistuu kevyesti ja vaivattomasti. Erona perinteisempään palvelimeen on se, että datan vastaanotto ja lähetys ei pysäytä koodin suorittamista. Tätä kutsutaan termillä non-blocking I/O. I/O toimii taustalla asynkronisesti (Kuva 3). Esimerkiksi PHP-palvelimelle koodin suoritus pysähtyy, kun odotellaan vastausta tietokannasta (blocking I/O). (Peltomäki, 2020)

Node toimii vain yhdellä säikeellä. Tämä tekee palvelinsovelluksesta kevyen ja säästää palvelimen resursseja. Raskaat prosessia vaativat pyynnöt hidastavat Node-sovellusta, minkä vuoksi Nodea ei suositella raskasta prosessointia vaativille sovelluksille. Node on parhaimmillaan datavetoisissa tiedonsiirroissa, sekä kun käyttäjiä on monia samanaikaisesti. Esimerkiksi PHP-palvelimella jokaiselle käyttäjälle pitää avata oma säe koodin suoritukseen. Samanaikaiset käyttäjät vaativat paljon resursseja PHP-palvelimelta, verrattuna Nodeen, joka käyttää vain yhtä säettä. (Peltomäki, 2020)

3.4 Asynkronisuus

Noden päätapahtumasilmukka (Kuva 3) on yksisäikeinen, mutta Node tukee rinnakkaisuutta tapahtumien ja asynkronisten metodien avulla. Päätasolla JavaScript-koodi suoritetaan aina pääsilmukassa eli yhdessä säikeessä. Koodia ei välttämättä suoriteta rivijärjestyksessä, vaan jos esimerkiksi edellisen rivin koodin suoritus on vielä kesken, voidaan silti aloittaa jo seuraavan rivin suorittaminen. Esimerkiksi käyttöliittymässä painetaan nappia, joka suorittaa palvelimella koodin, vaikka samanaikaisesti suoritetaan taustalla pidempi kestoista prosessia. (Peltomäki, 2020)

Kuva 3. Noden tapahtumasilmukka.



3.5 Moduulit

Moduulit ovat Noden sovelluskehityksessä tärkeitä, sillä itse Noden ydin on kevyt ja yksinkertainen. Moduulit tuovat tarvittavat lisäominaisuudet ja kehittämiseen nopeutta, sekä helppoutta. Nodeen on sisäänrakennettu moduuleita (W3schools, n.d.-a) ja NPM:n kautta pystyy lataamaan lisää. Lisäksi Nodella on helppoa tehdä omia moduuleita. Modulaarinen kehittäminen onkin suotavaa, että sovelluksen runko pysyy siistinä. (W3schools, n.d.-b)

Moduulit otetaan käyttöön require-määreellä tiedoston alussa, jonka jälkeen moduulin ominaisuuksia ja toimintoja pystyy käyttämään koodissa. Ohjelmakoodissa 4. Express-moduulin käyttöönotto. (Peltomäki, 2020)

Ohjelmakoodi 4. Express-moduulin käyttöönotto

```
const express = require('express')
```

4 Palvelinpuolen työkalut

Sovelluksen kehittämisessä käytetään yleensä erilaisia tekniikoita ja samassa projektissa voi olla useampikin kehittäjä. Projektinhallinnan kannalta on tärkeää, että kaikki tietävät työnkulun vaiheet ja käytettävät tekniikat. Kehittämisen ja projektinhallinnan helpottamista varten on suunniteltua erilaisia työkaluja. Tässä luvussa esitellään palvelinpuolella käytettäviä työkaluja, sovelluksia ja sovelluskehyksiä.

4.1 NPM

Node Package Manager eli NPM on maailman suurin ohjelmistorekisteri. NPM asentuu Noden mukana vakiona. Monet avoimen lähdekoodin kehittäjät käyttävät NPM:ää ohjelmistokirjastojen jakamiseen ja lainaamiseen. NPM:n avulla on helppo asentaa kirjastoja projektiin.

Node on hyvin pelkistetty systeemi ilman lisäosia eli kirjastoja. Kirjastojen avulla sovellusten tekeminen helpottuu huomattavasti. NPM:n avulla onnistuu Noden kirjastojen asennus, sekä riippuvuuksien asennus projektiin. Moduulin eli kirjaston asennus (Komento 1) tapahtuu komentorivillä.

Komento 1. Moduulin asennus NPM

```
$ npm install express
```

Npm:n tietoturva on ollut ajankohtainen aihe. Esimerkiksi vuonna 2021 julkaistiin suositusta kirjastosta tietoturva-aukko. Komponenttia oli ladattu noin 34 miljoonaa kertaa. Haavoittuvuus koski Systeminformation-kirjastoa, jonka tarkoituksena oli hakea tietoa laitteistosta ja sen prosesseista. Hyökkääjä pystyi antamaan järjestelmäkomentoja syöttämällä kirjastolle sopivia parametreja. Kirjaston haavoittuvuus korjattiin ja kehittäjien piti päivittää kirjasto uuteen versioon (Sharma, 2021).

Projektin käyttäessä monia riippuvuuksia on kehittäjän vaikea pysyä kärryillä riippuvuuksien yhteensopivuudesta sekä niiden tietoturvaongelmista. Vuonna 2018 NPM julkaisi riippuvuuksien ja haavoittuvuuksien hallintaan Audit-työkalun. Auditilla tarkasteltiin kirjastojen haavoittuvuudet,

yhteensopivuudet sekä päivitykset (NPM, n.d.). GitHub-versionhallintaa käyttävät saavat automaattisesti ilmoituksia tietoturvauhista ja riippuvuuksista. (GitHub, n.d.-a)

4.2 Express.js

Node tarjoaa vain peruspaketin HTTP-palvelimen rakentamiseen, mutta lisäosilla eli moduuleilla pystytään rakentamaan ammattimaisia palvelimia. Yleisin ja tunnetuin näistä moduuleista on Express. Express-moduulilla on erittäin helppo rakentaa Nodella kehitettävä HTTP-palvelin, myös REST Web Service-palvelut onnistuvat. Express on ilmainen MIT-lisenssillä julkaistu moduuli, jonka kehittäjä, T.J. Holowaychuk, kuvaili Express-kehystä minimaaliseksi. (Peltomäki, 2020)

Yksinkertainen Express-moduulilla tehty web-palvelin on vain muutama rivi koodia (Ohjelmakoodi 5). Ensimmäisellä rivillä otetaan käyttöön express-moduuli. Sijoitetaan app-muuttujaan express-sovellusta vastaava olio. Määritetään reititys polkuun ”/” eli kaikki palvelimelle tulevat GET-pyyntöihin vastataan ”Hello World!” -tekstillä. Viimeiset rivit kertovat HTTP-palvelimen kuuntelevan porttiin 3000 tulevia HTTP-pyyntöjä.

Ohjelmakoodi 5. Yksinkertainen HTTP-palvelin Express moduulilla

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000, () => {
  console.log(`Example app listening on port 3000`)
})
```

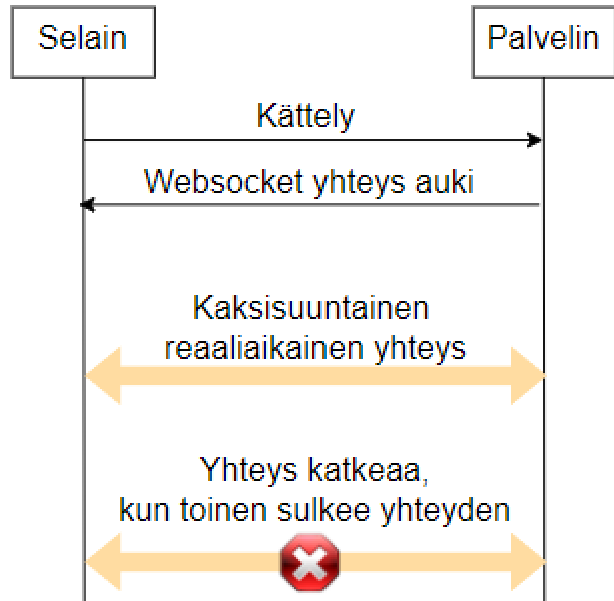
4.3 SocketIO

Reaaliaikaisen yhteyden toteuttamiseen Nodelle on suunniteltu socket.IO-moduuli. Selainpuolen ja palvelimen välinen kaksisuuntainen yhteys, joka muodostetaan käyttäen Websocket-yhteyttä.

TCP-protokollan päällä toimiva Websocket-teknologia on tekniikka kahdensuuntaiseen keskusteluun selaimen ja palvelimen välillä. Websocketin API-rajapinta on standardisoitu, joten kaikki selaimet tukevat sitä. HTTP-pyynnöllä selain kättelee palvelimen kanssa. Hyväksytyn kättelyn jälkeen palvelin avaa Websocket-yhteyden. Yhteyden ollessa auki selain sekä palvelin kumpikin voivat lähettää viestejä toisilleen. Kuvassa 4 esimerkkinä WebSocket-yhteyden elinkaari.

SocketIO-moduuli on yleisin kirjasto WebSocket-yhteyden toteuttamiseen. Palvelinpuolen koodi kirjoitetaan Nodella. Selainpuoli on HTML5/JavaScript toteutus. SocketIO:n toiminta perustuu eventteihin eli tapahtumiin. Selaimelle eli käyttäjälle luodaan Socket-olio ja oliolle luodaan haluttuja eventtejä. Eventtejä voidaan emitoida käyttäjältä palvelimelle, sekä toisinpäin. Emitit ovat kuin kutsuja tai käskyjä suorittaa tapahtumia. (Tuikka, n.d.)

Kuva 4. WebSocket-yhteys



4.4 Nodemon

Nodemon on hyvä ja erittäin hyödyllinen kehitysvaiheen moduuli. Nodemonin avulla palvelimen tallennetun muutoksen jälkeen palvelin käynnistyy uudelleen automaattisesti, eikä palvelinta tarvitse käynnistää uudelleen manuaalisesti. Tämä nopeuttaa, sekä tekee kehittämisen mukavammaksi. Package.json -tiedostoon määritellään Nodemonille NPM-skripti (Ohjelmakoodi

6), jonka avulla voimme käynnistää palvelimen kehitystilassa. Komento **$ npm run dev** käynnistää palvelimen kehitystilassa. (Nodemon, n.d.)

Ohjelmakoodi 6. NPM skriptit

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
},
```

4.5 Visual Studio Code

Visual Studio Code on avoimen koodin monialustainen tekstieditori. Visual Studio Code on saatavilla Windows-, Linux- ja MacOS-käyttöjärjestelmille. Tekstieditori tukee suosituimpia ohjelmointikieliä perusversiona. Ladattavilla laajennuksilla muokattavissa tarpeiden mukaan, niin tekstieditorin teemoja kuin uusia ohjelmointikieliä hyödyntäen. (Microsoft, 2022)

4.6 Versionhallinta GitHubilla

GitHub on versionhallintatyökalu, jonka avulla voidaan säilöä koodia ja koodin eri vaiheita. Kehityksen vaiheet voidaan nimetä, jotta niihin on helpompi palata tarvittaessa. Sovelluksen uusien ominaisuuksien käyttöönoton mennessä pieleen, voidaan palata aikaisempaan versioon. (GitHub, n.d.-b)

Versionhallintatyökalu helpottaa ryhmässä työskentelyä. Koodia on helppo jakaa, sekä henkilöitä voi kutsua projekteihin. Muiden tekemää koodia pystyy käyttämään ja kehittämään, jopa ilman fyysistä kontaktia toisiin kehittäjiin. Projekteissa pystyy antamaan kommentteja, vikailmoituksia ja kehitysideoita. Koodin kehityksen vaiheet näkevät kaikki projektiin osallistujat, mikä helpottaa projektiin tuntemista ja yhteistyötä. (GitHub, n.d.-b)

5 Chat-sovelluksen toteutus

Sovelluksen toteuttaminen aloitettiin tarvittavien työkalujen asennuksesta. Tämän jälkeen oli vuorossa itse Chat-ohjelman perustoiminnallisuuksien ja käyttöliittymän rungon suunnittelu. Ohjelmakoodia kirjoitettiin, niin että toiminallisuus piti saada toimimaan ilman virhettä, ennen uuden toiminallisuuden aloittamista. Ohjelmakoodin kirjoittamiseen käytettiin Visual Studio Code -tekstieditoria, joka oli tuttu ohjelmoinnin kursseilta.

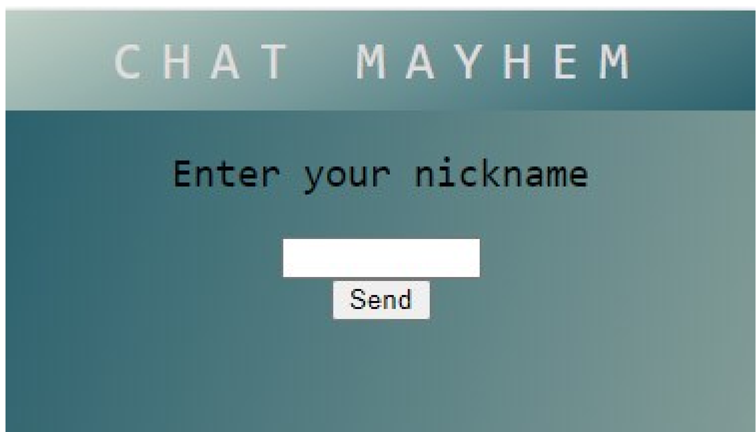
5.1 Asennukset

Kaikki käyttämäni ohjelmistot ovat ilmaisia ja ladattavissa ohjelmistojen verkkosivulta. Asennetavia ohjelmia sovelluksen toteuttamiseen oli vain kaksi, Visual Studio Code -tekstieditori (https://code.visualstudio.com/) ja Node.js-ympäristö (https://nodejs.org/en/). Lisäksi asensin GitHub Desktop -ohjelmiston (versionhallinnan graafinen työpöytäsovellus), sovelluksen etenemisen seuraamiseen, mutta tämä ei ole välttämätön sovelluksen toteutuksen kannalta. GitHub Desktop:n voi ladata osoitteesta: https://desktop.github.com/.

5.2 Toiminallisuus

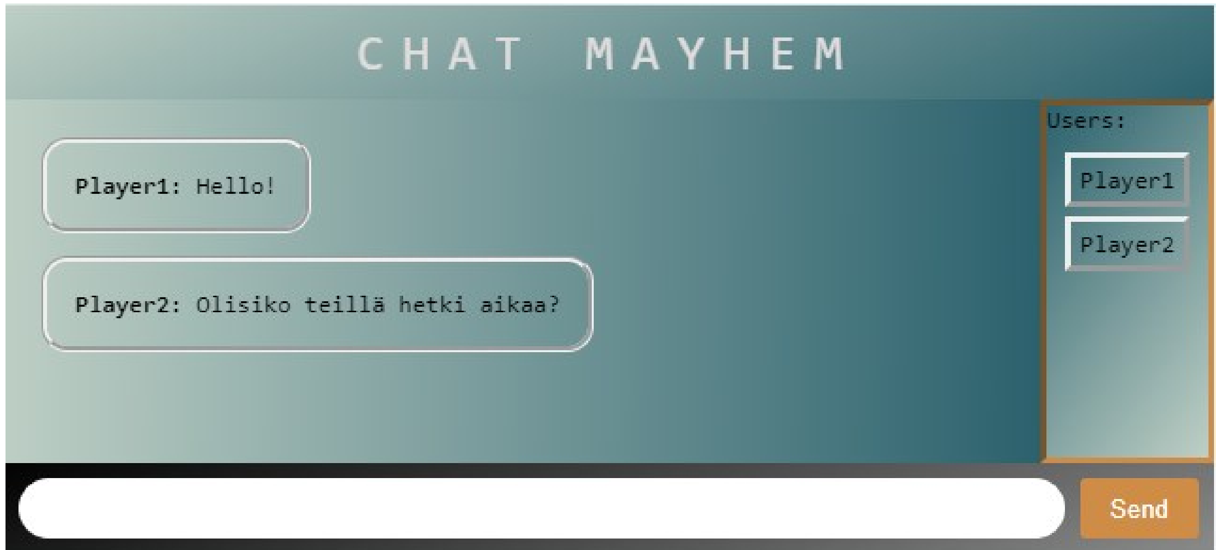
Ensin palvelin toimittaa käyttöliittymän selaimelle (Kuva 5). Käyttäjälle avautuu sovelluksen käyttöliittymä, jossa kysytään ensin käyttäjältä nimeä, jota halutaan käyttää chat-sovelluksessa. Käyttäjän yrittäessä sisään jo käytössä olevalla nimellä tulee varoitus, että ”Käyttäjänimi on jo käytössä”. Kirjautuessa käyttäjälle luodaan sessio ja sessiolle arvotaan id, joka tallennetaan käyttäjän selaimen muistiin. Käyttäjän päivitettyä sivu tai palatessa sivulle, käyttäjän ei tarvitse enää syöttää käyttäjänimeä. Palvelin tarkastaa selaimelta saatavan sessiotunnuksen(sessionID) ja palauttaa sen pohjalta oikean käyttäjänimen.

Kuva 5. Käyttäjänimen syöttäminen



Kirjautumisen jälkeen kirjautumisikkuna menee piiloon ja käyttäjälle aukeaa chat-näkymä (Kuva 6). Chat-ikkunan alareunassa on tekstinsyöttö ja lähetys nappi. Käyttäjät listataan oikeassa reunassa. Käyttäjä voi lähettää ja vastaanottaa viestejä päänäkymässä, sekä vastaanottaa privaattiviestejä.

Kuva 6. Chat-ikkuna

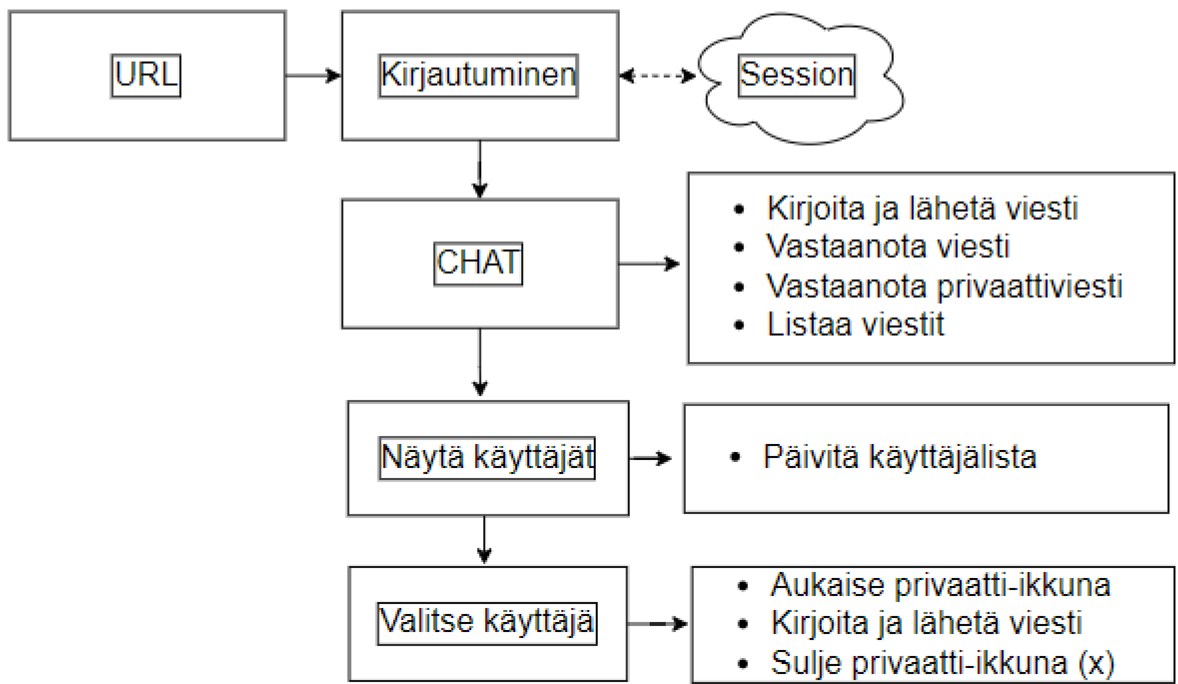


Käyttäjälistan jotakin käyttäjää klikatessa, aukeaa privaatti-ikkuna (Kuva 7). Vain klikatulle käyttäjälle voi laittaa privaattiviestin tässä ikkunassa. Käyttäjien keskeiset viestit näkyvät privaatti-ikkunassa.

Kuva 7. Privaatti-ikkuna

Sovelluksen toimintakaavio (Kuva 8) selventää tapahtumien järjestystä.

Kuva 8. Chat-sovelluksen toimintakaavio.

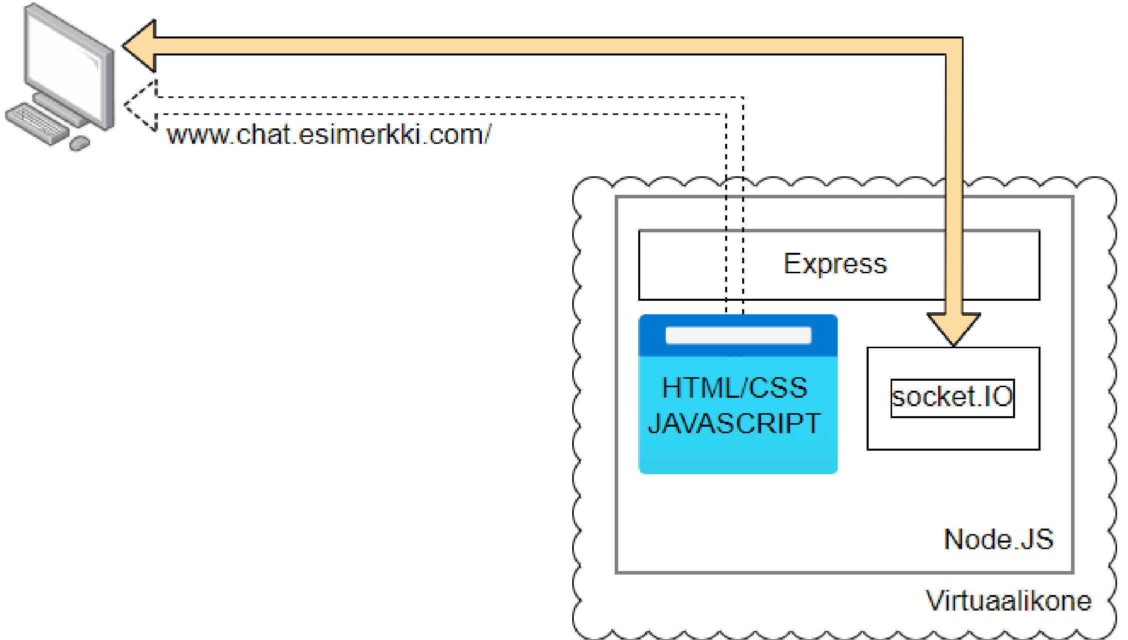


5.3 Sovelluksen rakenne

Node on käynnissä tietokoneella tai pilvessä virtuaalikoneella. Node taas suorittaa kirjoittamaani koodia. Noden sisällä tai ikään kuin pinnalla taas toimivat moduulit Express ja socketIO. Kuva 9. helpottaa hahmottamaan sovelluksen rakennetta ja toimintaa.

Selaimen osoiteriville kirjoitetaan palvelimen osoite. Selain lähettää palvelimelle pyynnön. Express-kehys hoitaa HTTP-pyynnön ja palvelin palauttaa vastauksena käyttöliittymän, index.html --tiedoston. Selaimella käyttäjä lähettää palvelimelle HTTP-pyynnön Websocket-yhteyden avaamiseksi, Express-kehys hoitaa pyynnön socket.IO-kehykseen, josta aukeaa kahdensuuntainen yhteys selaimen ja palvelimen välille.

Kuva 9. Sovelluksen rakenne.



5.4 Ohjelmakoodi

Luvussa esitellään projektin ohjelmakoodia sekä komentorivin koodeja. Komentorivin koodit alkavat $-merkillä. Projektin koko ohjelmakoodi löytyy liitteistä, sekä avoimena projektina GitHubista: https://github.com/Paavokal/oppari/tree/main/backend

5.4.1 Projektin aloitus

Node.js-projekti kannattaa aloittaa komennolla **$ npm init**, joka luo rungon projektin kansioon. Komennon jälkeen ohjelma kyselee projektiin liittyviä kysymyksiä: esimerkiksi projektin nimi, versio, käynnistyspiste sekä tekijä. Sopivin vastauksien jälkeen tuloksena on tiedosto nimeltä package.json. Tiedosto sisältää projektin määrittelyt ja riippuvuudet.

Tämän jälkeen asennetaan tarvittavat moduulikirjastot Express ja socket.IO. **$ npm install express socket.io**. Asennetaan myös Nodemon-kirjasto, kehityksen aikaiseksi riippuvuudeksi, **$ npm install --save-dev nodemon.** Komennon välissä parametri ”–save-dev” lisää nodemon-kirjaston kehityksen aikaiseksi riippuvuudeksi, jota ei olisi enää tuotantoversiossa. Tämän jälkeen package.json-tiedosto näyttää liitteen 2. ohjelmakoodin mukaiselta.

5.4.2 Express web-palvelin

Ensin luodaan ohjelman juureen tiedosto index.js. Tiedoston alkuun määritellään express-kirjasto, luodaan muuttuja app ja sijoitetaan siihen express-sovellusta vastaava olio (Ohjelmakoodi 7). HTTP-palvelin saa express-kehyksestä(app) tarvittavat funktiot.

Seuraavaksi määritellään sovellukselle reititys(*route*) polkuun ”/” eli tämä tapahtumankäsittelijä hoitaa sovelluksen juureen tulevia GET-pyyntöjä. Funktiolla on kaksi parametriä req(request) ja res(response) eli HTTP-pyynnön tiedot ja response on vastauksen määritelmä.

Ohjelmakoodin viimeiset rivit laittavat server-muuttujan HTTP-palvelimen kuuntelemaan porttiin 3001 tulevia HTTP-pyyntöjä.

Ohjelmakoodi 7. HTTP-palvelin Expressillä

```
const express = require('express');
const app = express();
const http = require('http');
const server = http.createServer(app);

//Tässä välissä loput koodit
```

```
server.listen(3001, () => {
  console.log('listening on *:3001');
});
```

5.4.3 Socket.IO

Socket.IO-kirjasto otetaan käyttöön tiedoston index.js alussa. Alustetaan uusi socketIO-instanssi välittämällä sille HTTP-palvelimen objekti server (Ohjelmakoodi 8).

Ohjelmakoodi 8. SocketIO:n käyttöönotto

```
const socketIo = require('socket.io')

const io = socketIo(server)
```

5.4.4 Käyttöliittymä

Käyttöliittymä luotiin työssä projektikansion sisälle public-kansioon, tiedoston nimeksi index.html. Käyttöliittymä on staattinen eli se ladataan vain kerran käyttäjälle. Tämän toiminallisuuden hoitaa Express-kirjaston middleware static. Ohjelmakoodi ohjaa kaikki GET-pyynnöt public-kansioon, josta tarjotaan käyttöliittymän sisältämä index.html -tiedosto (Ohjelmakoodi 9).

Ohjelmakoodi 9. Express static

```
app.use(express.static(__dirname + '/public'))
```

5.4.5 Kirjautuminen

Ennen socket-yhteyden avaamista tarkastetaan, tuleeko selaimen GET-pyynnön mukana sessionID(session numero). Session löytyessä se verrataan olemassa oleviin ja sieltä poimitaan käyttäjän tiedot (esim. Liite 3.). Käyttäjän syötettyä käyttäjänimi, lähetetään se palvelimelle tarkasteltavaksi, löytyykö sessioista samanniminen käyttäjä. Sessiosta saadut tiedot syötetään socket-olioon. Jos sessiota ei löydy, niin käyttäjälle luodaan uusi sessionID, sekä userID (käyttäjänumero). Nyt käyttäjän socket-yhteydellä on valmiina käyttäjänimi, sessionID ja userID, ennen socket-yhteyden luomista.

5.4.6 Socket-yhteys

Ohjelman vertailtua sessiot, avataan socket-yhteys palvelimen ja selaimen välille. Selaimelta tulee tapahtuma socket.connect(), koska selainpuolen socketin alustuksessa määriteltiin autoConnect-toiminto pois päältä eli automaattinen yhteyden muodostus. Vakiona automaattinen yhteyden muodostus on päällä. Palvelinpuolella kaikki toiminta tapahtuu serverin ollessa yhteydessä selainpuolen sockettiin, kuten ohjelmakoodissa Ohjelmakoodi 10 havainnollistetaan.

Ohjelmakoodi 10. Socket-yhteys

```
io.on('connection',(socket)=>{

//Tässä välissä tapahtuu kaikki toiminnot, kun socket-yhteys on auki.

  socket.on('disconnnect',() => {
  //Tämä tapahtuu, kun yhteys katkeaa
  }
}
```

5.4.7 Session

Palvelimen ja selaimen socket-yhteyden avauksen jälkeen, session tiedot tallennetaan palvelimelle ajon aikaiseen muistiin, sekä lähetetään session tiedot selaimelle. Socket.emit -komennolla (Ohjelmakoodi 11) lähetetään selainpuolelle eli käyttäjälle session tiedot. Selainpuolen koodista löytyy socket.on, jolla taas kuunnellaan emit-komennolla lähetettyjä samannimisiä viestejä.

Sessiot tallennetaan palvelimelle ajonaikaiseen muistiin, joten palvelimen uudelleenkäynnistyksessä sessiot katoavat. Sessiot tallennetaan sessionStore-nimiseen Map-kokoelmaan.

Ohjelmakoodi 11. Session tallennus ja lähetys

```
sessionStore.saveSession(socket.sessionID, {
    userID: socket.userID,
    username: socket.username,
    connected: true,
})
socket.emit("session", {
    sessionID: socket.sessionID,
    userID: socket.userID,
    username: socket.username
})
```

5.4.8 Käyttäjät

Socketin eli käyttäjän avattua yhteys palvelimelle, palvelin hakee sessionStore-kokoelmasta kaikkien käyttäjien tiedot ja tallentaa sen users-listaan (Ohjelmakoodi 12). Users-lista lähetetään kaikille palvelimella komennolla io.emit(”users”, users).

Ohjelmakoodi 12. Käyttäjien haku

```
sessionStore.findAllSessions().forEach((session) => {
    if(session.connected === true) {
        users.push({
        userID: session.userID,
        username: session.username,
        connected: session.connected,
        })
    }
})
io.emit("users", users)
```

5.4.9 Viesti

Palvelin kuuntelee selaimelta lähetettyä ”chat message”-nimistä lähetystä. Palvelin jakaa tämän viestin kaikille palvelimella, komennolla io.emit(…)(Ohjelmakoodi 13). Viestin alkuun muokataan socketin käyttäjänimi (socket.username) ja loppuun itse viesti (msg). Palvelimeen yhteydessä olevat selaimet kuuntelevat tätä lähetystä socket.on(”chat message”…) ja suorittavat sen sisällä olevat toiminnot.

Ohjelmakoodi 13. Viestin lähettäminen.

```
socket.on('chat message', msg => {
    io.emit('chat message',`<b>${socket.username}</b>: ${msg}`)
})
```

5.4.10 Privaattiviestit

Palvelin kuuntelee selaimelta tulevaa lähetystä nimeltä ”private_message”. Sen parametreina ovat viestin vastaanottajan käyttäjänumero (toUser), sekä viesti (msg). Lähetys toimitetaan vain kahdelle käyttäjälle käyttäen koodia socket.to(…).socket.to(…).emit().

5.4.11 Palvelimen käynnistys

Kehitysvaiheessa palvelin käynnistettiin Nodemonia hyödyntäen komennolla ”**npm run dev”**. Tämä onnistui, koska package.json -tiedoston skripteihin (Ohjelmakoodi 14) oli lisätty käsky **dev**. Palvelimen normaali käynnistäminen tapahtui komennolla **$ npm start**.

Ohjelmakoodi 14. NPM-skriptit.

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js",
    "dev": "nodemon index.js"
  }
```

6 Tulokset

Toteutuksessa saatiin aikaan yksinkertainen chat-sovellus käyttäen palvelinpuolen Node-alustaa, sekä chat-sovelluksella sopivia Express- ja socketIO-moduulikirjastoja. Kirjastojen avulla ohjelmakoodin kirjoittaminen oli tehty todella vaivattomaksi. Lisäksi tekijöiden sivuilla oli paljon esimerkkejä erilaisista ohjelmista.

Sovelluksen kehittäminen jäi kesken, koska aikaa oli toteutukseen vain reilu viikko. Jatkokehitysideana oli lisätä chat-sovellukseen toiminallisuuksia, kuten viestien tallentaminen ja hakeminen erillisestä tietokannasta. Sovelluksen kehitys jatkuu tämän opinnäytetyöprosessin jälkeen.

Vaikeuksia projektissa tuottivat eniten selainpuolen CSS- ja JavaScript-koodit, joihin kului suunniteltua enemmän aikaa. Selainpuolen koodiin meni huomattavasti enemmän aikaa verrattuna palvelinpuolen koodin. Projektin aihe ei ollut tekijälle entuudestaan tuttu, joten opetteluun ja testailuun meni paljon aikaa.

7 Johtopäätökset ja pohdinta

Opinnäytetyön tutkimuskysymyksissä mietittiin mikä on web-sovellus ja miten se eroaa tavallisesta nettisivusta, sekä mikä on web-sovelluksen tietoturvan laita. Lisäksi mietittiin mikä on Node.js (Node) ja mitä kirjastoja käytetään reaaliaikaisen chat-sovelluksen toteuttamiseen. Työn toiminnan osuudessa toteutettiin yksinkertainen chat-sovellus, jossa keskityttiin enemmän palvelinpuolen koodiin, mutta kirjoitettiin pakolliset koodit myös selainpuolelle toiminnallisuuksien esittelyyn.

Web-sovelluksesta saatiin yleiskatsaus, jossa lukija ymmärtänee, mikä on web-sovellus ja miten se eroaa perus nettisivuista eli staattisista sivuista. Lisäksi käytiin läpi web-sovelluksien yleisiä tekniikoita ja ohjelmointikieliä. Web-sovelluksien tietoturvaosuudessa huomattiin, kuinka paljon web-sovelluksiin kohdistuu uhkia verrattuna esimerkiksi työpöytäsovelluksiin. Web-sovelluksien kehittäessä onkin syytä ottaa huomioon yleisimmät uhat.

Node.js-alustasta saatiin kattavampi selvitys. Perehdyttiin myös chat-sovelluksen toteutukseen käytettyihin moduulikirjastoihin. Lisäksi käsiteltiin esimerkin projektissa käytettyjä palvelinpuolen työkaluja. JavaScriptistä tekijä tiesi vain yleispiirteet, mutta ei sen syvällisempää kokemusta. Node valikoitui aiheeksi sen suosion takia. Tekijällä ei ollut aikaisempaa kokemusta Nodesta. Teorian pohjalta saatiin kuitenkin yksinkertainen, mutta erittäin toimiva chat-sovellus.

Tekijä sisäisti Noden alkeet, toimintaperiaatteen sekä onnistui kehittämään sillä toimivan sovelluksen. Jos työn tekisi uudestaan, niin aikaa käytettäisiin enemmän projektiin ja varsinkin selainpuoleen. Lisäksi projekti suunniteltaisiin paremmin. Tämän pohjalta tekijä aikoo jatkaa Node-projekteja tulevaisuudessa ja kehittää JavaScript osaamistaan.

8 Yhteenveto

Opinnäytetyön prosessi on ollut mielenkiintoinen ja opettava kokemus. Tekijällä ei ollut aikaisempaa kokemusta palvelinpuolen JavaScriptistä tai Node-alustasta. Päätavoitteena oli oppia uutta tekniikkaa ja tehdä sen pohjalta projekti. Projekti oli melko yksinkertainen ja tekijä olisikin halunnut kehittää sitä lisää, mutta tässä on syytä ottaa huomioon koko opinnäytetyön prosessiin käytettävä aika. Projektin tekemiseen meni arviolta viidennes ajasta, mitä käytettiin raportin kirjoittamiseen. Raportin kirjoittaminen oli työläin osuus. Opinnäytetyöstä sain suunnitelmien mukaisen, paitsi projektista jäi puuttumaan erillinen tietokanta.

Työn keskeisin osuus oli Node-alustaan tutustuminen. Nodella web-sovelluksen kehittäminen on tehty helpoksi noviisillekin. Netistä löytyy paljon ohjeita ja esimerkkejä itseopiskeluun. Projektissa tehtiin yksinkertainen chat-sovellus. Lopputulokseen tekijä on tyytyväinen. Projektissa yllätti selainpuolen haasteellisuus. Tekijältä meni huomattavasti enemmän aikaa selainpuolen koodiin, kuin palvelinpuoleen.

Tekijä sai opinnäytetyön prosessista hyvän käsityksen ja seuraava työ menisi sujuvammin. Node-alustasta opittiin perusteet ja tekijä aikoo jatkossa opiskella palvelinpuolen kehittämistä. Chat-sovelluksen jatkokehitysideana on viestien tallentaminen tietokannan avulla.

Lähteet

GitHub. (n.d.-a). *About code scanning - GitHub Docs*. Retrieved March 3, 2022, from https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/about-code-scanning

GitHub. (n.d.-b). *Getting started with GitHub - GitHub Docs*. Retrieved February 16, 2022, from https://docs.github.com/en/get-started

Karhu Helsinki Oy. (n.d.). *Mitä web-kehitys on? | Karhu Helsinki Oy*. Retrieved February 28, 2022, from https://www.karhuhelsinki.fi/blogi/mita-web-kehitys

Martin. (2021). *Difference between Website and Web Application (Web App)*. B. https://www.guru99.com/difference-web-application-website.html

Microsoft. (2022). *Documentation for Visual Studio Code*. https://code.visualstudio.com/docs

Mozilla. (2022). *JavaScript language resources - JavaScript | MDN*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources

Nodemon. (n.d.). *nodemon*. Retrieved February 21, 2022, from https://nodemon.io/

NPM. (n.d.). *About npm | npm Docs*. Retrieved February 2, 2022, from https://docs.npmjs.com/about-npm

OWASP. (n.d.). *Introduction - OWASP Top 10:2021*. Retrieved February 15, 2022, from https://owasp.org/Top10/A00_2021_Introduction/

Peltomäki, J. (2020). *Node.js Web-palveluiden ohjelmointi*. BoD, Helsinki.

Second Nature Security. (2018). *KRIITTISIMMÄT HAAVOITTUVUUDET WEB-SOVELLUKSISSA*. www.2ns.fi

Sharma. (2021). *Heavily used Node.js package has a code injection vulnerability*. https://www.bleepingcomputer.com/news/security/heavily-used-nodejs-package-has-a-code-injection-vulnerability/

Tuikka, T. (n.d.). *Nodejs-sovelluskehitys*. Retrieved February 2, 2022, from https://tiko.jamk.fi/~tuito/websk20/nodesk/index.html

W3schools. (n.d.-a). *Node.js Built-in Modules*. Retrieved February 21, 2022, from https://www.w3schools.com/nodejs/ref_modules.asp

W3schools. (n.d.-b). *Node.js Modules*. Retrieved February 21, 2022, from https://www.w3schools.com/nodejs/nodejs_modules.asp

W3schools. (2022a). *CSS Introduction*. https://www.w3schools.com/css/css_intro.asp

W3schools. (2022b). *What is HTML*. https://www.w3schools.com/whatis/whatis_html.asp

w3techs. (n.d.). *Usage Statistics of JavaScript as Client-side Programming Language on Websites, February 2022*. Retrieved February 21, 2022, from https://w3techs.com/technologies/details/cp-javascript

Wirfs-Brock, A., & Eich, B. (2020). JavaScript: The first 20 years. *Proceedings of the ACM on Programming Languages*, *4*(HOPL), 189. https://doi.org/10.1145/3386327

Liite 1: Aineistonhallintasuunnitelma

Työn aikana tehdään muistiinpanoja tekijän toimesta, sekä pidetään päiväkirjaa GitHub palvelussa. Muistiinpanot ovat tekijän hallussa toteuttamiseen asti, minkä jälkeen muistiinpano hävitetään. Päiväkirja on GitHubissa säilössä kirjautumisen takana. Päiväkirjaa säilytetään ainakin vuoden verran opinnäytetyön valmistumisesta.

Liite 2: Ohjelmakoodi: package.json

```
{
  "name": "chat-sovellus",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
      "start": "node index.js",
      "dev": "nodemon index.js",
      "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Paavo Kalliala",
  "license": "MIT",
  "dependencies": {
      "express": "^4.17.1",
      "socket.io": "^4.4.1"
  },
  "devDependencies": {
      "nodemon": "^2.0.7"
  }
}
```

Liite 3: Palvelinpuolen ohjelmakoodi: index.js

```
const express = require('express')
const app = express()
const http = require('http')
const server = http.createServer(app)
const PORT = process.env.PORT || 3001
const socketIo = require('socket.io')
const crypto = require('crypto')
const randomId = () => crypto.randomBytes(8).toString("hex")
require("dotenv").config()

const { InMemorySessionStore } = require("./sessionStore");
const sessionStore = new InMemorySessionStore();

const io = socketIo(server)

app.use(express.static(__dirname + '/public'))


// MIDDLEWARE: USERNAME SOCKET HANDSHAKE
io.use((socket, next) => {
    //CHECK AVAILABLE SESSION
    const sessionID = socket.handshake.auth.sessionID
    if (sessionID) {
      const session = sessionStore.findSession(sessionID)
      if (session) {
        socket.sessionID = sessionID
        socket.userID = session.userID
        socket.username = session.username
        return next()
      }
    }

    //CHECK IF USERNAME ALREADY TAKEN
    sessionStore.findAllSessions().forEach((session) => {
        if(session.username === socket.handshake.auth.username) {
            return next(new Error("username already taken"))
        }
    })
    const username = socket.handshake.auth.username
    if (!username) {
      return next(new Error("invalid username"))
    }
    socket.sessionID = randomId()
    socket.userID = randomId()
    socket.username = username
    next()
  })


//CLIENT CONNECTION
io.on('connection',(socket)=>{
    socket.join(socket.userID)
    const users = []


    //SAVE SESSION
    sessionStore.saveSession(socket.sessionID, {
        userID: socket.userID,
```

```
        username: socket.username,
        connected: true,
    })
    socket.emit("session", {
        sessionID: socket.sessionID,
        userID: socket.userID,
        username: socket.username
    })


    //LOGGING ALL EVENTS
    socket.onAny((event, ...args) => {
        console.log(event, args)
    })

    //USER JOIN / update userlist
    sessionStore.findAllSessions().forEach((session) => {
        if(session.connected === true) {
            users.push({
            userID: session.userID,
            username: session.username,
            connected: session.connected,
            })
        }
    })
    console.log(users)
    io.emit("users", users)

    //CHAT-MESSAGE
    socket.on('chat message', msg => {
        io.emit('chat message',`<b>${socket.username}</b>: ${msg}`)
    })
    //PRIVATE MESSAGE
    socket.on('private_message', (toUser, msg) => {
        console.log(`private message: "${msg}" from:${socket.userID}
to:${toUser}`)
        io.to(toUser).to(socket.userID).emit('private_message', {
            msg: `<b>${socket.username}:</b> ${msg}`,
            from: socket.userID,
            to: toUser
        })
    })

    socket.on('new message', () => {
        io.to(socket.userID).emit('users', users)
    })

    //USER DISCONNECT
    socket.on('disconnect', () => {
        //DISCONNECTED USER connected:false
        sessionStore.saveSession(socket.sessionID, {
            userID: socket.userID,
            username: socket.username,
            connected: false,
        });
        console.log('User: ' + socket.userID + ' has left')
        io.emit('user_disconnect', socket.userID)
    })
})
    server.listen(PORT, () => {
    console.log('listening on *:3001')
})
```

Liite 4: Selainpuolen ohjelmakoodi (index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Chatti</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" type="text/css" href="index.css">
  </head>
  <body>
      <div id="app">

        <div id="header">
            <div id="site-title"> CHAT MAYHEM </div>
        </div>

        <div id="chat-main">
            <!--LOGIN window-->
            <div id="loginWindow">
                <p>Enter your nickname</p>
                <p id="nickError"></p>
                <form id="setNickname">
                  <input size="35" id="nickname" maxlength="10"> </input>
                  <button>Send</button>
                </form>
            </div>
            <!--CHAT window-->
            <div id="chat">
                <div id="privateWindow">
                    <div id="privateHeader">
                        <div id="privateTitle"></div>
                        <button id="privateExit">X</button>
                    </div>
                    <ul id="privateMessages"></ul>
                </div>
                <div id="messageWindow">
                            <ul id="messages"></ul>
                </div>
                <div id="userWindow">
                    <div>Users:</div>
                    <ul id="users"></ul>
                </div>
            </div>

        </div>

        <div id="footer">
            <form id="form" action="">
                <input id="input" autocomplete="off"
/><button>Send</button>
            </form>
            <form id="privateForm" action="">
                <input id="privateInput" autocomplete="off"
/><button>Send</button>
            </form>
        </div>
      </div>

      <script src="/socket.io/socket.io.js"></script>

      <script>
```

```
const socket = io({autoConnect:false})
console.log(socket)

const sessionID = window.localStorage.getItem('sessionID')
if(sessionID) {
  socket.auth = { sessionID }
  socket.connect()
}


const form = document.getElementById('form')
const input = document.getElementById('input')
const chat = document.getElementById('chat')
const setNickname = document.getElementById('setNickname')
const loginWindow = document.getElementById('loginWindow')
const nickInput = document.getElementById('nickname')
const messageWindow = document.getElementById('messageWindow')
const userList = document.getElementById('users')
const privateWindow = document.getElementById('privateWindow')
const privateHeader = document.getElementById('privateHeader')
const privateTitle = document.getElementById('privateTitle')
const privateForm = document.getElementById('privateForm')
const privateExit = document.getElementById('privateExit')
const nickname = ''
var toUser = ''
privaUser = []
newMessage = []
curUser = []


userUpdate = (users) => {
  console.log(newMessage)
  const usersConnected = users.map(u => u)
      userList.innerHTML= ''
      usersConnected.forEach(element => {
          const item = document.createElement('li')
          item.innerHTML = element.username
          // item.style.fontWeight = 'normal'
          if(newMessage.includes(element.userID)){
              // item.style.background = 'white'
              item.id = 'blink'
          }
          item.addEventListener('click', () => {
              privateMessage(element)
          })
          userList.appendChild(item)
      })


}



//SESSION ON
socket.on("session", ({ sessionID, userID, username }) => {
  // attach the session ID to the next reconnection attempts
  socket.auth = { sessionID }
  // store it in the localStorage
  window.localStorage.setItem('sessionID', sessionID)
  // save the ID of the user
  socket.userID = userID
  chat.style.display = 'grid'
```

```
    form.style.display = 'flex'
    loginWindow.style.display = 'none'
    document.title = `Chatti - ${username}`
  })



  // SET NICKNAME EVENTHANDLER
  setNickname.addEventListener('submit', (e) => {
        e.preventDefault()

        if(nickInput.value){
            socket.auth = { username:nickInput.value }
            socket.connect()
      }
  })

  //CHAT-MESSAGE EVENTHANDLER
  form.addEventListener('submit', (e) =>{
    e.preventDefault()

    if(input.value.startsWith('/w')){
        const words = input.value.split(' ')
        const toUser = words[1]
        const msgToUser = input.value.substr(input.value.indexOf(
                    ' ', input.value.indexOf(' ') + 1) +1)

        console.log(toUser + ' -> ' +msgToUser)
        socket.emit('private_message', toUser, msgToUser)
        input.value = ''
    }

    else if(input.value) {
        socket.emit('chat message', input.value)
        input.value = ''
    }
  })

  //SEND PRIVATE MESSAGE
  privateForm.addEventListener('submit', (e) =>{
    e.preventDefault()
    const msg = privateInput.value
    socket.emit('private_message', toUser, msg)
    privateInput.value = ''
  })

  //CLOSE PRIVATE WINDOW
  privateExit.addEventListener('click', (e) =>{
    e.preventDefault()
    privateWindow.style.display = 'none'
    messageWindow.style.display = 'block'
    privateForm.style.display = 'none'
    form.style.display = 'flex'
    toUser = ''
  })

//OPEN PRIVATE WINDOW
const privateMessage = (element) => {
    privateWindow.style.display = 'block'
    privateTitle.innerHTML = element.username
    messageWindow.style.display = 'none'
    privateForm.style.display = 'flex'
    form.style.display = 'none'
```

```
            toUser = element.userID
            console.log('priva ikkuna auki')
            console.log(toUser)
            console.log(privaUser)

            privateMessages.innerHTML=''
            privaUser.forEach(pu => {
                //private to self
                if(element.userID === socket.userID){
                    if (pu.to === element.userID && pu.from ===
element.userID) {
                        var item = document.createElement('li')
                        item.innerHTML = pu.msg
                        privateMessages.appendChild(item)
                    }
                }
                else if (pu.to === element.userID || pu.from ===
element.userID) {
                var item = document.createElement('li')
                item.innerHTML = pu.msg
                privateMessages.appendChild(item)
                privateWindow.scrollTo(0, privateWindow.scrollHeight)
                }
            })
            newMessage = newMessage.filter(u => u !== element.userID)
            userUpdate(curUsers)

          }

          //USERS UPDATE
          socket.on('users', (users) => {
              console.log('päivitetään käyttäjät')
              console.log(users)
              curUsers=users
              userUpdate(curUsers)

          })

          //CHAT MESSAGE
          socket.on('chat message', (msg) => {
            var item = document.createElement('li')
            item.innerHTML = msg
            messages.appendChild(item)
            messageWindow.scrollTo(0, messageWindow.scrollHeight)
          })

          //PRIVATE MESSAGE
          socket.on('private_message', (data) => {
            console.log(data.from +' private message tulee: ' + data.msg)
            var item = document.createElement('li')
            item.innerHTML = data.msg
            privateMessages.appendChild(item)
            privateWindow.scrollTo(0, privateWindow.scrollHeight)
            privaUser.push(data)
            if(data.from !== socket.userID){
                newMessage.push(data.from)
            }
                userUpdate(curUsers)
          })

          socket.on('user_disconnect', (userID) => {
                console.log('poistetaan ' + userID)
```

```
                curUsers.splice(curUsers.findIndex(u => u.userID ===
userID), 1)
                userUpdate(curUsers)
          })


          //CONNECT ERRORS
          socket.on("connect_error", (err) => {
            if (err.message === "invalid username") {
                console.log('Invalid username')
            }

            if (err.message === "username already taken") {
                chat.style.display = 'none'
                form.style.display = 'none'
                loginWindow.style.display = 'flex'
                alert('username already taken')
                console.log('username already taken')
            }
          })

      </script>
  </body>
</html>
```