Tampereen ammattikorkeakoulu

React-sovellukselle backendin toteuttaminen

Lauri Tikka

OPINNÄYTETYÖ
Huhtikuu 2025

Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

TIKKA, LAURI
React-sovellukselle Backendin touteuttaminen

Opinnäytetyö 29 sivua
Huhtikuu 2025

Opinnäytetyön taustalla oli luoda backend-järjestelmä web-sovellukselle, josta sellainen puuttuu. Kyseinen web-sovellus on onnenpyörä, joka on toteutettu React-kirjastolla. Onnenpyöräsovelluksen käyttäjät voivat luoda listan pelaajista, joista onnenpyörä valitsee yhden satunnaisesti.

Työn tarkoituksena oli suunnitella ja toteuttaa toimiva backend-järjestelmä, joka mahdollistaa datan tallentamisen ja hakemisen palvelimelta. Tavoitteena oli kehittää ohjelmointirajapinta (API) ja tietokanta, jotka yhdessä toteuttavat backend-järjestelmän.

Tämän saavuttamiseksi API toteutettiin Node.js-ympäristössä hyödyntäen Express-sovelluskehystä. Tietokannaksi tälle backendille valittiin PostgreSQL. API ja tietokanta ajetaan docker-kontteina backendin käyttöönoton ja hallinnan helpottamiseksi.

Työn tuloksena valmistui toimiva backend-ratkaisu, joka toteuttaa onnenpyöräsovelluksen vaatimukset. Backend tarjoaa skaalautuvamman perustan projektin jatkokehitykselle. Yhteenvetona voidaan todeta, että nykyaikaiset teknologiat mahdollistavat modulaarisen ja ylläpidettävän backend-järjestelmän rakentamisen kaikenlaisiin sovelluksiin.

Asiasanat: api, node.js, react, tietokanta

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

TIKKA, LAURI:
Implementing a Backend for a React Application

Bachelor's thesis 29 pages
April 2025

The background for this thesis was to implement a backend for a web application which lacks one. This web application is a fortune wheel implemented using React framework. Users of this application are able to create a list of players one of which this fortune wheel randomly picks.

The purpose of this thesis was to design and implement a working backend which allows storing and retrieving data from a server. The goal was to build an application programming interface (API) and a database that together form the backend system for this project.

To achieve this, the API for the backend was implemented in Node.js environment with the help of Express framework. The database chosen for this project was PostgreSQL. Both API and database run as docker containers to help with deployment and management.

The project resulted in a functional backend solution that meets the fortune wheel application’s requirements. The backend provides a more scalable foundation for further development of this project. In conclusion, modern technologies make it possible to build a modular and maintainable backend system for all kinds of applications.

Key words: api, node.js, react, database

SISÄLLYSLUETTELO

LYHENTEET JA TERMIT

API	Ohjelmointirajapinta.
Backend	Sovelluksen palvelimella suoritettava osa, joka toimii taustajärjestelmänä.
Docker	Työkalu sovelluskonttien hallintaan.
Express	Web-sovelluskehys.
JWT	JSON Web Token, usein autentikoinnissa käytetty tapa siirtää tietoa kahden järjestelmän välillä.
Node.js	Selaimen ulkopuolinen Javascript-ajoympäristö.
PostgreSQL	Suosittu relaatiotietokanta.
REST	Tilaton kommunikointirajapinta

1 JOHDANTO

Opinnäytetyön tavoitteena on kehittää backend aikaisempaa opiskelijaprojektia varten. Kyseinen opiskelijaprojekti on React-sovelluskehyksellä toteutettu onnenpyörä. Tällä onnenpyörällä käyttäjä pystyy lisäämään listaan pelaajia, joista animoidulla onnenpyörällä arvotaan yksi voittaja. Onnenpyörä on kuitenkin toteutettu ainoastaan web-sivuna, josta puuttuu täysin taustajärjestelmä kuten API ja tietokanta.

Tämän työn tavoitteena on jatkokehittää projektia luomalla API ja tietokanta tätä onnenpyöräsovellusta varten. Näiden avulla onnenpyörän käyttäjä kykenee tallentamaan useampia listoja, joita onnenpyörän kanssa voi käyttää. Koska alkuperäisestä projektista puuttuu tietokanta, siinä ei ole kykyä tallentaa pelaajalistaa pidempiaikaisesti. Lista on luotava uudestaan, kun sovellus käynnistetään uudelleen. Tietokannan myötä tätä ongelmaa ei ole.

Tietokantaan ja API:in tullaan toteuttamaan myös käyttäjien kirjautuminen. Tämä mahdollistaa sen, että useampi käyttäjä voi käyttää onnenpyörää omilla listoillaan.

2 KÄYTETYT TEKNOLOGIAT

2.1 Node.js

Javascriptin suosio on sen historian aikana levinnyt web-selaimen ulkopuolelle moneen muuhunkin käyttöön. Yksi esimerkki tästä on Node.js, jota tässä työssä tullaan käyttämään. Node.js-ajoympäristö mahdollistaa Javascript-koodin ajamisen web-selaimen ulkopuolella (Node.js. 2025). Sillä on monia suosittuja käyttökohteita, kuten palvelimet, web-sovellukset ja komentorivityökalut.

Onnenpyöräsovellus on toteutettu typescript-kielellä, joka on käytännössä laajennus Javascriptille. Tämän vuoksi on luonnollinen valinta toteuttaa API Javascriptillä, sillä se yhtenäistää projektissa käytettyjä teknologioita, jolloin kokonaisuuden ylläpito ja kehitys on sujuvampaa.

2.2 Express

Express on Node.js:lle luotu sovelluskehys, jonka tarkoituksena on helpottaa web-sovellusten tekemistä (Express. 2025). Se on suuressa suosiossa RESTful APIen kehityksessä ja sitä tullaan hyödyntämään myös tämän projektin parissa. API:n totetutus onnistuisi ilman ylimääräistä sovelluskehystäkin suoraan Node.js:n päälle. Expressillä kuitenkin saadaan koodista paljon suoraviivaisempaa ja luettavampaa. Kyseinen sovelluskehys sisältää valmiita paljon valmiita ominaisuuksia, jotka helpottavat ja nopetuttavat kehitystyötä.

2.3 PostgreSQL

Usein web-sovellukset, jotka sisältävän verkkosivun ja API:n, tarvitsevat myös tietokantaa datan pidempiaikaiseen tallentamiseen. Tämän työn tietokannaksi on valikoitunut PostgreSQL. PostgreSQL on relaatiotietokanta, joka tunnetaan sen suorituskyvystä, laajennettavuudesta ja muista ominaisuuksista, joita ei välttämättä kaikista relaatiotietokannoista löydy. Laajennusmahdollisuuksia, mitä PostgreSQL:stä löytyy ovat muun muassa:

- Funktiot ja proseduurit
- Laaja tuki ohjelmointikielille
- Yhdistäminen muihin tietokantoihin ja tietovirtoihin
- Laaja valikoima lisäosia

Tietokanta, joka tullaan rakentamaan, tulee sisältämään yksinkertaista taulukkodataa. Tämän vuoksi sillä ei ole suurta merkitystä, mikä tietokanta työhön valitaan. Mikä vain yleisesti käytössä oleva tietokanta toimisi yhtä hyvin tähän tarkoitukseen. PostgreSQL kuitenkin valikoitui sen laajennusmahdollisuuksien takia. Tämä tuo mahdollisuuksia projektin jatkokehitykselle tietokannan kannalta, ilman, että sitä täytyy kokonaan vaihtaa toiseen.

2.4 JSON Web Token

API tulee sisältämään käyttäjiä, joten käyttäjien autentikoinnin toteutukselle on tarve. JWT, eli Javascript Web Token on tapa siirtää tietoa kahden järjestelmän välillä (Auth0. 2025). Esimerkiksi palvelin pystyy tallentamaan asiakkaan selaimeen kryptografisesti allekirjoitetun merkkijonon. Tämä merkkijono voi sisältää palvelimen asettamia tietoja, kuten käyttäjänimen, sähköpostin tai käyttäjän oikeuksia. Seuraavaksi asiakas voi lähettää HTTP-pyynnön yhteydessä tämän merkkijonon. Koska palvelin on hoitanut kyseisen merkkijonon allekirjoittamisen, se pystyy tarkistamaan, ettei sen sisältämiä tietoja ole muutettu.

JWT:n etuna on se, että se on tilaton. Tämä tarkoittaa käytännössä sitä, että palvelimen ei tarvitse tallentaa tietoa aktiivisista istunnoista. Perinteisemmässä istunnonhallintamallissa palvelimen täytyy pitää muistissa aktiiviset istunnot. JWT:n kanssa palvelimen ei tarvitse tallentaa tilatietoja istunnosta. Palvelimen tehtävänä on tarkistaa tokenin allekirjoitus, jotta tiedetään, että tokenin tietoja ei olla muutettu.

2.5 Docker

Docker on ohjelmistoalusta, joka mahdollistaa sovellusten rakentamisen ja julkaisun helposti. Docker käyttää niin kutsuttuja kontteja, jotka sisältävät sovelluksen tarvitsemat kirjastot, työkalut, lähdekoodin ja ajoympäristön. Nämä kontit hyödyntävät Linux-ytimen tarjoamia ominaisuuksia, joka tekee niistä virtuaalikoneita paljon kevyempiä. Virtuaalikoneet ovat virtualisoituja tietokoneita, jotka käyttäytyvät oikean kokonaisen tietokoneen tavoin. Niiden ajaminen on huomattavasti raskaampaa ja käyttöönotto vaivalloisempaa kuin konttien.

Ilman Dockeria täytyisi asentaa kaikki riippuvuudet omaan ajoympäristöön. Jos käyttäjä haluaa vain ajaa projektin eikä kehittää sitä, täytyisi hänen asentaa kehitystyökaluja ja riippuvuuksia käyttöjärjestelmäänsä. Tästä voisi myös seurata versio-ongelmia eri riippuvuuksien kanssa. Dockerin kanssa tämä kaikki on hoidettu automaattisesti, kun kontteja laitetaan pystyyn.

3 API

3.1 API yleisesti

API toteutetaan REST-rajapintana. REST on rajapintojen toteuttamiseen tarkoitettu malli, jossa rajapinnan tarjoamat tiedot esitetään resursseina. Oleellista REST-rajapinnoissa on tilattomuus, joka tarkoittaa sitä, että palvelin ei tallenna asiakkaan kyselyistä mitään tilatietoja. Rajapintaa käyttäessä on lähetettävä kaikki tarvittava tieto jokaisessa kyselyssä. Koska API toteutetaan web-sovelluksena, sen resurssit esitetään URL-osoitteen polkuina.

Käytetty API:n toiminto määrittyy HTTP-metodin avulla. Näitä metodeja ovat esimerkiksi GET resurssien hakemiseen, DELETE resurssien poistamiseen ja POST resurssien lisäämiseen. Resurssi, jota HTTP-kutsussa käsitellään, määritetään URL-osoitteen polun avulla. API toteuttaa siis CRUD-operaatiot päivitystoimintoa lukuun ottamatta. CRUD (Create, Read, Update, Delete) määrittää neljä perustoimintoa tiedon hallintaan, luomisen, lukemisen, päivittämisen ja poistamisen (Pang 2021). Työssä käytetty data on tarpeeksi yksinkertaista, että päivitystoiminnolle ei ole tarvetta.

APIssa on kaksi eri päätepistettä. Ensimmäinen on /api/users käyttäjien hallintaan ja toinen on /api/lists käyttäjien listojen hallintaan. Taulukko 1. sisältää listojenhallintaan liittyvät API-polut.

TAULUKKO 1. API-polut listojen halintaa varten.

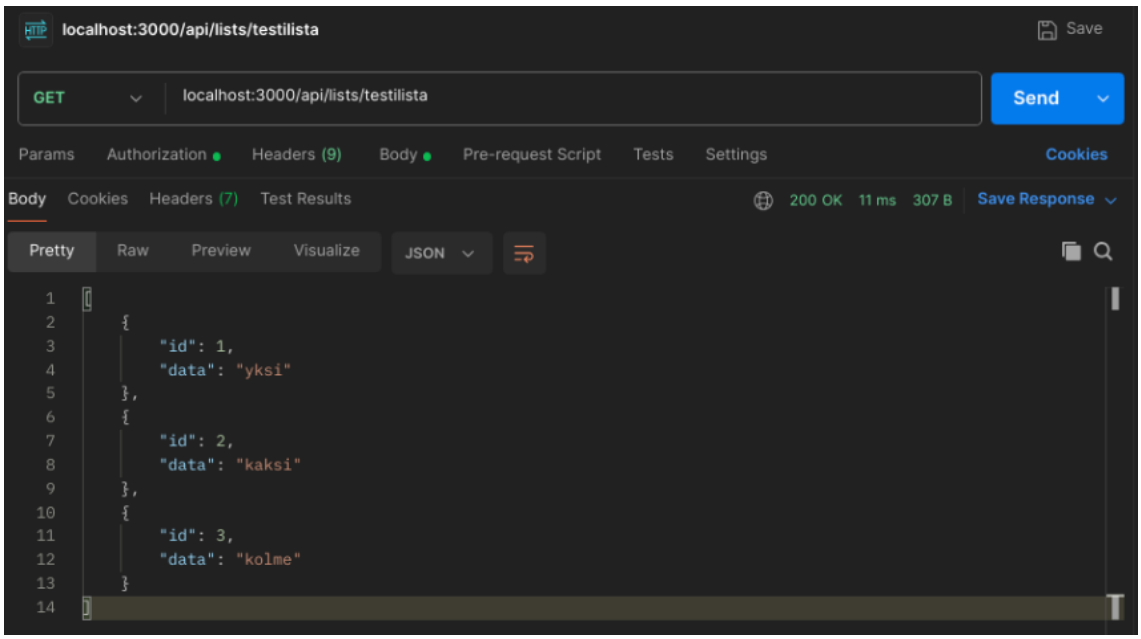
HTTP-metodi	Polut
GET	GET /api/lists/ - Palauta kaikki käyttäjän listat. GET /api/lists/{listName} – Palauta tietty käyttäjän lista.
POST	POST /api/lists/{listName} – Luo uusi lista. POST /api/lists{listName}/{item} – Lisää

	uusi kohde listaan.
DELETE	DELETE /api/lists/{listName} – Poista lista. DELETE /api/lists/{listName}/{name} – Poista kohde listasta.

TAULUKKO 2. API-polut käyttäjienhallintaa varten

HTTP-metodi	Polut
GET	GET /api/lists/ - Palauta kaikki käyttäjän listat. GET /api/lists/{listName} – Palauta tietty käyttäjän lista.
POST	POST /api/lists/register – Rekisteröi uusi käyttäjä. POST /api/lists/login – Kirjaudu sisään käyttäjänä.

Kirjautumis- ja rekisteröitymistiedot lähetetään HTTP-pyynnön sisällössä JSON-muodossa. JSON on tiedostomuoto, joka on helppoa luettavaa ihmiselle. Se sai alkunsa Javascript-standardin mukana, mutta on kuitenkin täysin kielestä riippumaton ja laajentunut yleiseen käyttöön. API:n vastaukset käyttävät myös JSON:ia. Kuvassa 1. on esimerkki API:n lähettämästä vastauksesta, kun käyttäjän tiettyä listaa API:lta haetaan.



KUVA 1. API-kysely Postman-työkalulla

3.2 Toteutus

3.2.1 Polut ja reitittimet

API:n pääpolut on määritetty projektin aloitustiedostossa index.js (kuva 2). Koodissa listapolkujen ja käyttäjäpolkujen hallinta on määritetty omissa Express-reitittimissään (router). Jos käyttäjän pyytämä polku ei vastaa toista näistä poluista, käyttäjälle lähetetään vastaus tästä 404 HTTP-koodin kanssa.

```
const listsRouter = require("./routes/lists.router");
const userRouter = require("./routes/user.router");

app.use("/api/lists", listsRouter);
app.use("/api/users", userRouter);

app.use((req, res) => {
  res.status(404).json({ error: "Endpoint not found" });
});
```

KUVA 2. Pääpolkujen määritys

Tarkemmat polut ja niiden käsittelijäfunktiot ovat määritetty omassa tiedostossaan (kuva 3). Reitittimelle on määritetty lista polkuja, joita se sisältää. Esimer-

kiksi listan lisääminen tapahtuu polussa ’/api/users/:ListName’ POST-metodilla. Tämän kutsun käsittely on määritelty lists.controller-tiedoston createNewList-funktiossa. Näille poluille on lisäksi määritetty verifyToken niminen middleware-funktio, joka hoitaa JWT autentikoinnin varmistamisen. Tämä tarkoittaa sitä, että käyttäjän täytyy olla onnistuneesti kirjautunut, jotta se pystyy näitä polkuja käyttämään.

```
const express = require("express");
const router = express.Router();
const listsController = require("../controllers/lists.controller");

const verifyToken = require("../middleware/jwt");

// all list interactions needs to be verified by a JWT
router.get("/", verifyToken, listsController.getAllLists);
router.get("/:listName", verifyToken, listsController.getList); // get list
router.post("/:listName", verifyToken, listsController.createNewList); // create new list
router.post("/:listName/:itemName", verifyToken, listsController.addItemToList); // add item to a list
router.delete("/:listName", verifyToken, listsController.deleteList); // delete an existing list
router.delete("/:listName/:itemName", verifyToken, listsController.removeItemFromList); // delete item from a list

module.exports = router;
```

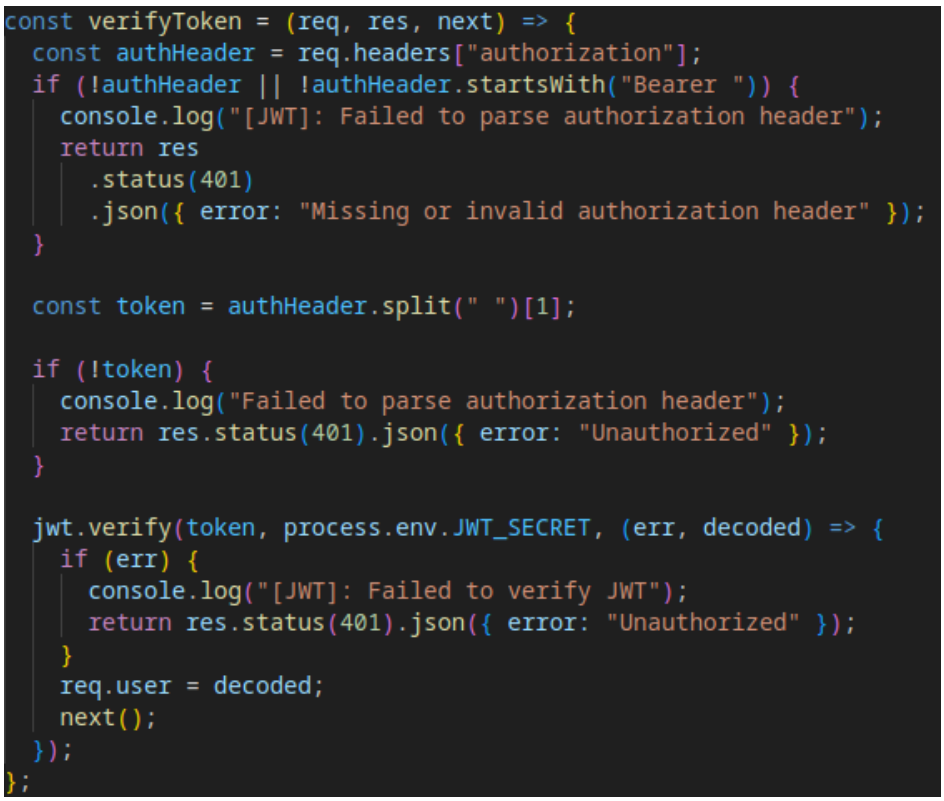
KUVA 3. Listapolkujen reititin

Käyttäjienhallintaan liittyvät polut on määritetty omassa tiedostossaan, joka sisältää toiminnallisuudet kirjautumiseen ja rekisteröitymiseen. Nämä polut eivät vaadi JWT:tä, mutta onnistuneen kirjautumisen tapauksessa ne palauttavat käyttäjälle JWT:n, jota käyttämällä se voi hallita omia listojaan.

3.2.2 JWT-autentikointi

Autentikointia vaativien API-polkujen käyttö on toteutettu middleware-funktiona. Middleware-funktiot ottavat käyttäjän lähettämän HTTP-pyynnön ja käsittelevät sitä, ennen kuin ne antavat pyynnön eteenpäin seuraavalle funktiolle.

Tässä tapauksessa autentikoinnin hoitava middleware-funktio lukee HTTP-pyynnöstä authorization nimisen otsikkokentän ja koittaa varmistaa sen. Jos varmistus ei onnistu niin käyttäjälle palautetaan vastaus, jossa kerrotaan autentikoinnin epäonnistuneen. Muuten tallennetaan JWT:stä luetut tiedot ja ohjelman suoritus siirretään seuraavalle funktiolle.

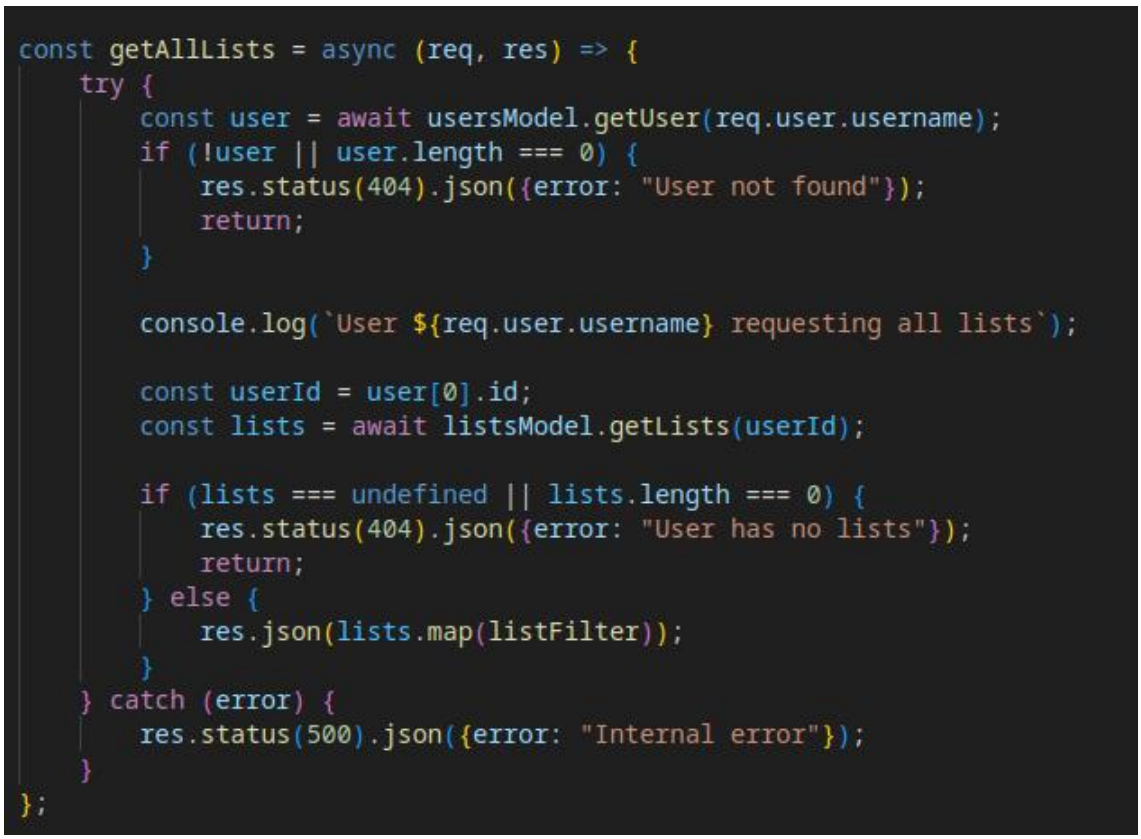


```
const verifyToken = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  if (!authHeader || !authHeader.startsWith("Bearer ")) {
    console.log("[JWT]: Failed to parse authorization header");
    return res
      .status(401)
      .json({ error: "Missing or invalid authorization header" });
  }

  const token = authHeader.split(" ")[1];

  if (!token) {
    console.log("Failed to parse authorization header");
    return res.status(401).json({ error: "Unauthorized" });
  }

  jwt.verify(token, process.env.JWT_SECRET, (err, decoded) => {
    if (err) {
      console.log("[JWT]: Failed to verify JWT");
      return res.status(401).json({ error: "Unauthorized" });
    }
    req.user = decoded;
    next();
  });
};
```

KUVA 4. Autentikoinnin toteutus

3.2.3 Kontrollerifunktiot

Kontrollerifunktiot sisältävät varsinaisen logiikan käyttäjän lähettämän pyynnön käsittelyyn. Nämä funktiot lähettävät kutsuja tietokannalle ja päättävät millaisilla vastauksilla käyttäjän pyynnöille vastataan takaisin. Kuvaan 4 on otettu esimerkiksi käyttäjän listojen hakeminen. Kontrollerifunktiot hoitavat muun muassa: kirjautumistietojen ja salasanojen hallinnan sekä HTTP-vastauksien muodostamisen.



```
const getAllLists = async (req, res) => {
    try {
        const user = await usersModel.getUser(req.user.username);
        if (!user || user.length === 0) {
            res.status(404).json({error: "User not found"});
            return;
        }

        console.log(`User ${req.user.username} requesting all lists`);

        const userId = user[0].id;
        const lists = await listsModel.getLists(userId);

        if (lists === undefined || lists.length === 0) {
            res.status(404).json({error: "User has no lists"});
            return;
        } else {
            res.json(lists.map(listFilter));
        }
    } catch (error) {
        res.status(500).json({error: "Internal error"});
    }
};
```

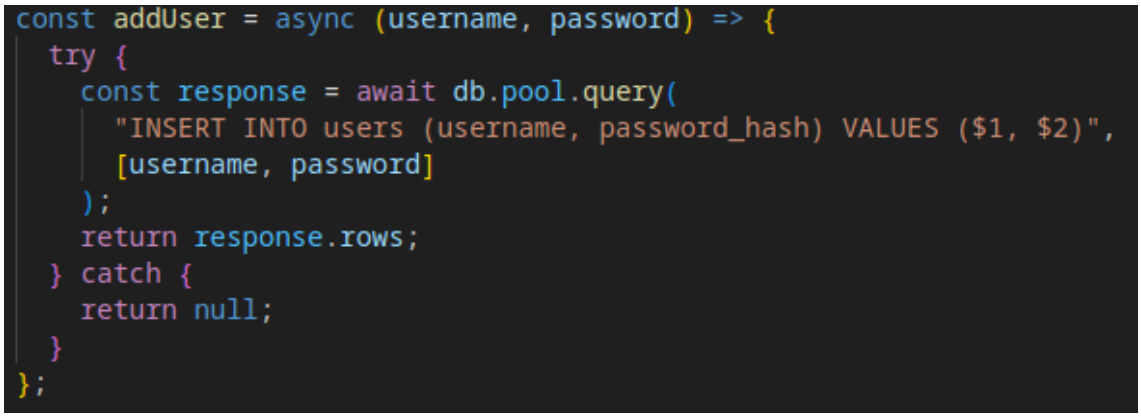
Kuva 5. getAllLists-funktio

Kuvasta näkee, kuinka ensiksi yritetään hakea tietokannasta käyttäjää JWT:n sisältämän käyttäjänimen perusteella. Jos käyttäjä löytyy, niin haetaan kaikki käyttäjän omistamat listat. Jos käyttäjällä ei ole listoja, vastaukseksi lähetetään tieto asiasta HTTP-vastauskoodilla 404. Muuten käyttäjälle lähetettään listat JSON-muodossa.

3.2.4 SQL-kyselyt

Viimeisessä vaiheessa, API:n täytyy lähettää varsinaiset tietokantakyselyt tietokannalle SQL-kielellä. Nämä ovat koottu omiin funktioihinsa, joita kontrollereiden on helppoa hyödyntää. Nämä funktiot ottavat yhteyden tietokantaan sen osoite- ja kirjautumistiedoilla ja palauttavat kyselyn tulokset, jos virheitä ei satu.

Yhteys tietokantaan on hoidettu yhteyspoolin avulla, joka pitää tietokantayhteyttä auki. Tämän hyötynä on se, että kun tietokannalle lähetetään jatkuvasti uusia kyselyitä, niin yhteyttä ei tarvitse jatkuvasti sulkea ja avata uudelleen. Jatkossa seuraavat kyselyt voidaan suorittaa saman avoinna olevan yhteyden kautta.



```
const addUser = async (username, password) => {
  try {
    const response = await db.pool.query(
      "INSERT INTO users (username, password_hash) VALUES ($1, $2)",
      [username, password]
    );
    return response.rows;
  } catch {
    return null;
  }
};
```

KUVA 6. Käyttäjän lisääminen tietokantaan.

4 TIETOKANTA

4.1 Tietokannan pystyttäminen

Tietokannan pystyttäminen on tehty mahdollisimman helpoksi. Kaikki tarvittava tieto on tallennettuna tiedostoihin, jolloin tietokannan pystyttäminen onnistuu vain yhdellä dockerin komennolla. Docker Composen tiedostoon on määritelty kontteihin liittyvät konfiguraatiot ja init.sql-tiedosto on skripti, jolla luodaan tietokantaan taulukot.

4.1.1 Docker Compose

Docker Compose on työkalu, jonka avulla voidaan määritellä ja ajaa useampia docker-kontteja kerralla. Nämä kontit ja niiden konfiguraatiot määritellään docker-compose.yaml tiedostossa, joka on YAML-muodossa. YAML on helppoluettava tiedostomuoto, jossa eri osiot määritellään sisennyksien avulla, hieman kuten Python-ohjelmointikielessä.

```yaml
version: '3.9'

services:
  db:
    image: postgres
    restart: always
    shm_size: 128mb
    ports:
    - "127.0.0.1:5432:5432"
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGES_DB: ${POSTGRES_DB}
    volumes:
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    env_file:
      .env

  adminer:
    image: adminer
    restart: always
    ports:
      - 8081:8080
```

KUVA 7. Docker-compose.yml

Kuvassa 7 on tietokannan kanssa käytetty Docker Compose -konfiguraatio. Services-osion alle on määritetty kaksi eri palvelua, db ja adminer. Adminer on web-pohjainen hallintatyökalu tietokannoille. Näistä luodaan omat kontit niiden omien määritysten mukaan.

- Image kertoo, mitä Docker-kuvaa käytetään.
- Restart tarkoittaa, että kontti käynnistetään uudelleen virheen sattuessa.
- Ports määrittää, mikä kontin ulkopuolelle näkyvä portti liitetään kontin sisällä olevaan porttiin.
- Environment-lohkossa on määritelty ympäristömuuttujia.
- Volumes määrittää tiedostojärjestelmä liitoksia kontin ulkopuolelta kontin sisälle.
- Env_file kertoo mistä tiedostosta ympäristömuuttujat luetaan.

Tarkemmin kuvailtuna tässä konfiguraatiossa saadaan dockerin sisäinen portti 5432, jota tietokanta käyttää, näkymään samassa portissa kontin ulkopuolelle. Tietokannan kontin tiedostojärjestelmään tehdään liitos sijaintiin ’/docker-entrypoint-init.db.d/init.sql’, johon liitetään tiedosto init.sql kontin isäntäjärjestelmästä.

4.1.2 Init.sql

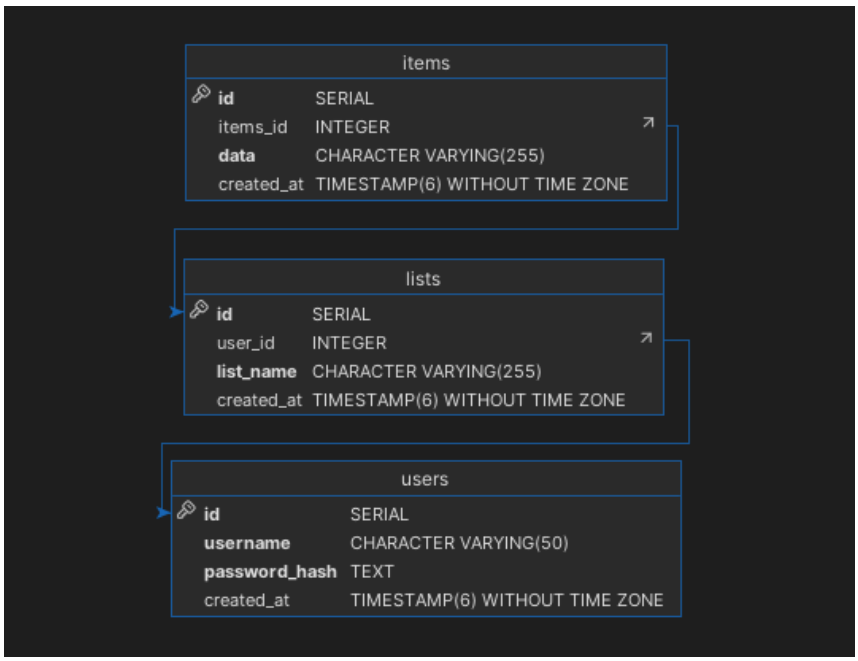
Sen jälkeen, kun tietokanta on saatu pystytettyä niin tietokantaan täytyy luoda taulukoita, jotta sinne voidaan tallentaa tietoa. Tämä on hoidettu automaattisesti konttien luomisen vaiheessa init.sql-tiedostossa. Kyseinen tiedosto on SQL-skripti, joka kopioidaan PostgreSQL:n hakemistoon, josta se etsii alustuksessa ajettavia skriptitiedostoja.

```
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS lists (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    list_name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    UNIQUE(user_id, list_name)
);

CREATE TABLE IF NOT EXISTS items (
    id SERIAL PRIMARY KEY,
    items_id INTEGER REFERENCES lists(id) ON DELETE CASCADE,
    data VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT NOW()
);
```

KUVA 8. Init.sql

4.2 Tietokannan rakenne

Kuvasta 9 nähdään taulut, jotka tietokantaan on luotu. Käyttäjät on tallennettu users nimiseen tauluun, joka sisältää ID:n, käyttäjänimen ja salatun salasanan. Taulukkoon lists on taas tallennettu kaikki listat, niiden nimet ja omistajat. Tämä taulukko sisältää viittauksen users-taulukkoon, jolloin tiedetään, kuka minkäkin listan omistaa. Viimeisenä taulukkona on items, joka sisältää tekstidataa ja viittauksen listaan. Tämän viittauksen avulla tiedetään, mihin listaan mikäkin listan kohde kuuluu.



KUVA 9. Tietokannan taulukot

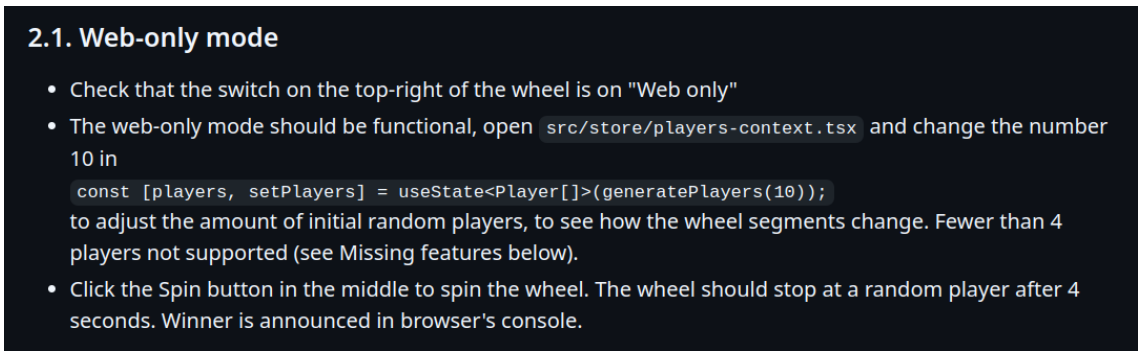
5 BACKENDIN KÄYTTÖÖNOTTO

5.1 Onnenpyörään tutustuminen

Tässä luvussa otamme luomamme backendin käyttöön. Aluksi täytyy selvittää, miten onnenpyöräsovellus luo ja hallitsee käyttämäänsä dataa. Sen jälkeen korvaamme kyseisen osan onnenpyörän koodista omilla funktioilla, jotka hallitsevat dataa API:n avulla. Tämä saattaa vaatia pieniä muutoksia web-sivun ulkonäköön, mutta koska tämän työn aiheena on backendin toteutus, niin pidämme ne mahdollisimman yksinkertaisina.

5.1.1 README.md

Onnenpyörään tutustuessa, projektin ohjeet antavat jo hyvin suuntaa sille, missä kyseisen sovelluksen käyttämä pelaajalista luodaan. Tiedostossa README.md mainitaan tiedosto players-context.tsx, josta löytyy generatePlayers-funktio. Tämä todennäköisesti viittaa siihen, että tällä funktiolla luotaisiin onnenpyörän käyttämä lista, josta pelaaja arvotaan. Tämän tiedoston nimi viittaisi siihen, että siinä on määritelty konteksti sovelluksen käyttämälle datalle.



KUVA 10. Onnenpyörän README.md

5.1.2 React-konteksti

Tiedosto players-context.tsx sisältää määritelmät kontekstille. React-sovelluksissa konteksti on tapa hallita sovelluksen sisältämää dataa ja tilaa

(React.dev. 2025). Kontekstin avulla React-komponenttien ei tarvitse siirtää dataa ylemmiltä komponenteilta alemmille, vaan kaikki komponentit voivat hakea ja päivittää sovelluksen dataa tämän kontekstin avulla. Näin ollen säästytään monimutkaiselta datan siirtelyltä komponentilta toiselle.

Onnenpyörässä sovelluksen konteksti on määritetty tiedostossa players-context.tsx. Kuvassa 11 näkyy pelaajalistalle määritetty konteksti. Tämä konteksti sisältää seuraavat asiat:

- Players-lista, johon varsinaiset pelaajat tallennetaan. Lista alustetaan generatePlayers-funktion avulla.
- AddPlayer-funktio, jonka avulla listaan lisätään pelaajia.
- RemovePlayer-funktio, jonka avulla listasta poistetaan pelaaja

```
const PlayersContextProvider = ({ children }: Props) => {
  const [players, setPlayers] = useState<Player[]>(generatePlayers(10));

  const addPlayer = (player: Player) => {
    setPlayers((prevPlayers) => [...prevPlayers, player]);
  };

  const removePlayer = (id: string) => {
    setPlayers((prevPlayers) =>
      prevPlayers.filter((player) => player.id !== id)
    );
  };

  const contextValue: PlayersContextObj = {
    players: players,
    addPlayer: addPlayer,
    removePlayer: removePlayer,
  };

  return (
    <PlayersContext.Provider value={contextValue}>
      {children}
    </PlayersContext.Provider>
  );
};
```

KUVA 11. Pelaajalistan konteksti

Tässä on lyhykäisyydessään osa sovelluksesta, joka täytyy korvata uudella backendilla. Sen sijaan, että kontekstin funktiot käsittelevät pelkästään players-listaa, niin niiden tulee hakea ensiksi pelaajat API:n kautta.

5.2 Autentikointi

Listojen hallintaan liittyvät API-kutsut tarvitsevat autentikointia. Tämä tarkoittaa sitä, että onnenpyörän on jollain tapaa hallittava ja muistettava onko käyttäjä kirjautunut sisään sekä tähän liittyvä token-arvo. Tämä token tullaan tallentamaan evästeenä selaimeen.

Jotta säästytään liian monimutkaiselta koodilta, tämän autentikoinnin tila on hyvä pitää yhdessä paikassa. Tämä saadaan toteutettua luomalla oma konteksti autentikoinnin hallintaan. Tämä autentikointikonteksti tulee sisältämään seuraavat toiminnot ja tiedot:

- IsAuthenticated-muuttuja, joka kertoo, onko käyttäjä kirjautunut sisään vai ei
- Username, joka tallentaa käyttäjän nimen
- Login-funktio, joka tallentaa API:n antaman tokenin evästeisiin ja asettaa isAuthenticated ja username muuttujat
- Logout-funktio, joka tyhjentää evästeet ja asettaa sopivat arvot isAuthenticated ja username muuttujille

5.3 API-kutsut

Nyt, kun pelaajalistan hallintaan käytetään API:a, täytyy aiemmat pelaajalistan päivitykset korvata sopivilla API-kutsuilla. Tarkastellaan, miten päivitetty versio onnenpyörästä lisää pelaajalistaan uuden pelaajan ja näyttää tämän päivitetyn listan.

Projektiin on luotu uusi hakemisto nimeltä “api”, johon on määritetty funktiot, jotka lähettävät varsinaiset kutsut API:lle. Koska onnenpyörä käsittelee vain yhtä listaa, pelaajalistat, johon se lisää ja poistaa pelaajia, niin on tarvinnut vain luoda kolme funktioita: fetchList, addToList ja removeFromList. Nimensä mukaan fetchList pyytää API:lta tietyn listan, addToList lisää uuden kohteen listaan ja removeFromList poistaa valitun kohteen listasta.

```
import { API_URL, COOKIE_TOKEN } from "../Definitions";

const getAuthHeaders = () => ({
  "Content-Type": "application/json",
  authorization: `Bearer ${localStorage.getItem(COOKIE_TOKEN)}`,
});

export const addToList = async (list: string, name: string) => {
  try {
    const response = await fetch(
      `${API_URL}/api/lists/${list}/${encodeURIComponent(name)}`,
      {
        method: "POST",
        headers: getAuthHeaders(),
      }
    );

    return await response.json();
  } catch (error) {
    console.error(`Failed to add ${list}:`, error);
    throw error;
  }
};
```

KUVA 12. Kohteen lisääminen listaan API:lla

Kuvan 12 funktio lähettää API:lle kutsun, joka lisää määritettyyn listaan määritetyn kohteen. Tämä funktio välitä API:n vastauksesta, vaan palauttaa sen suoraan JSON-muodossa. Tämä mahdollistaa sen, että virhetilanteet voidaan hallita tarkemmin muualla sovelluskoodissa.

Päivitetty pelaajakonteksti käyttää edellä esitettyjä API-funktioita avukseen. Sen sijaan, että kontekstin players-listaa päivitettäisiin suoraan käyttäjän syöttämillä arvoilla, lähetetään ensiksi API:lle pyyntö listan päivittämisestä ja jos se onnistuu, päivitetään pelaajakontekstin lista API:n palauttamalla uudella listalla.

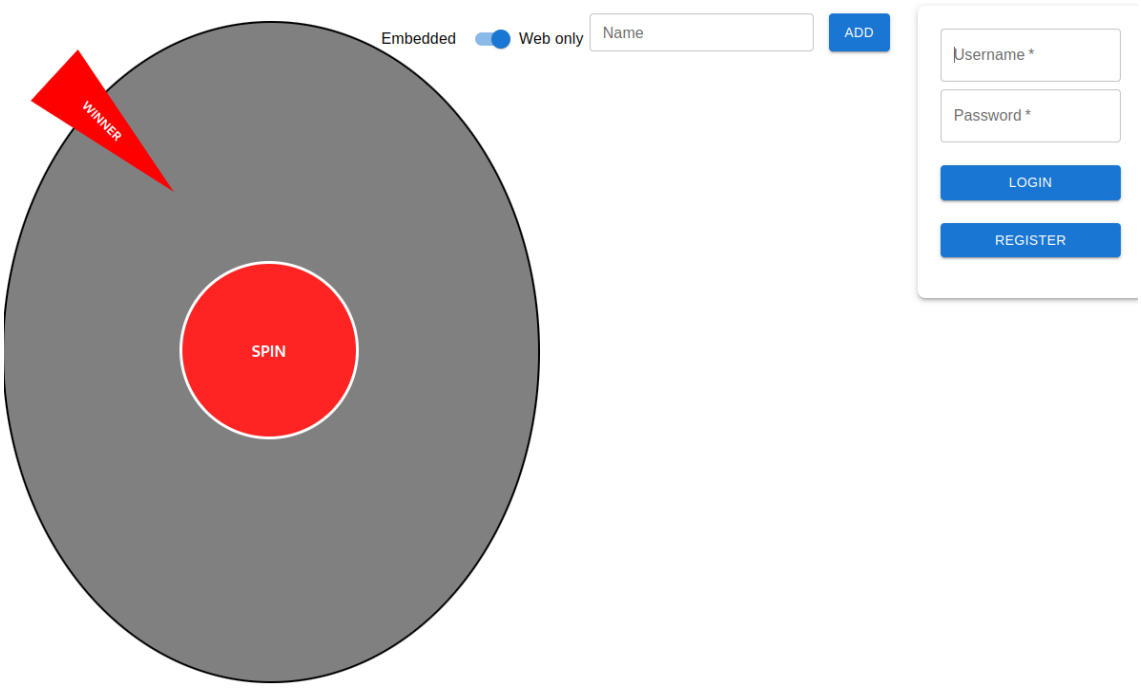
```
const transformPlayersData = (
  data: { id: number; data: string }[]
): Player[] => {
  return data.map(
    ({ id, data }: { id: number; data: string }): Player => ({
      id: id.toString(),
      name: data,
    })
  );
};

const addPlayer = async (player: Player) => {
  addToList("players", player.name)
    .then((data) => {
      if (data.error) {
        toast.error(data.error);
        return;
      } else {
        setPlayers(transformPlayersData(data));
      }
    })
    .catch((error) => console.log("Failed to add player: " + error));
};
```

KUVA 13. Onnenpyörän pelaajalistan päivitys

Ensiksi API:lle lähetetään kutsu players-listan päivityksestä käyttäjän syöttämällä player.name-arvolla. Jos tämä kutsu onnistuu, päivitetään pelaajalista API:n vastauksella. Jos API-kutsussa tapahtuu virhe, se näytetään käyttäjälle notifikaationa toast-kirjaston avulla. Pelaajalistan päivittämisessä käytetään apufunktiota transformPlayersData. Se sovittaa API:n palauttaman datan Player-datatyyppiin sopivaksi.

5.4 Login-komponentti

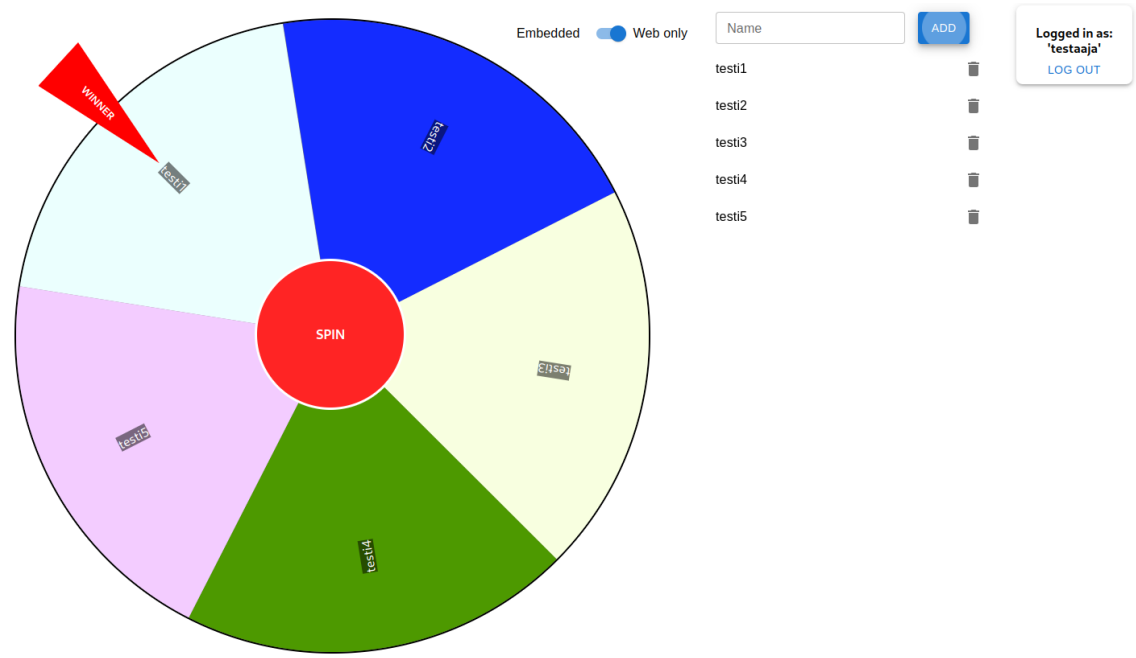
Alkuperäisessä onnenpyörän versiossa ei ollut mahdollisuutta kirjautua käyttäjätilille. Tämä tarkoittaa sitä, että luotua backendia varten täytyi kehittää yksinkertainen React-komponentti kirjautumista ja rekisteröintiä varten.



KUVA 14. Onnenpyörä kirjautumiskomponentin kanssa.

Tämä komponentti lähettää API:lle joko kirjautumis- tai rekisteröintipyynnön riippuen siitä, kumpaa nappia käyttäjä painaa. Login-nappi lähettää käyttäjätiedot API:n polkuun ‘/api/users/login’ ja register-nappi lähettää tiedot polkuun ‘/api/users/register’. Jos pyyntö onnistuu, niin tallennetaan API:n palauttama token selaimen evästeisiin käyttäen autentikointikontekstia.

Sisään kirjautunut käyttäjä näkee kirjautumiskomponentin sijaan tiedon omasta käyttäjänimestään ja napin uloskirjautumiseen. Uloskirjautuminen hoidetaan myös autentikointikontekstin avulla, joka poistaa tokenin evästeistä ja päivittää kontekstin tilan.



KUVA 15. Käyttäjä kirjautuneena sisään

6 POHDINTA

Työn tavoitteena oli jatkokehittää onnenpyöräsovellusta, joka oli aikaisempi opiskelijaprojekti. Onnenpyörälle toteutettiin backend, joka koostui API:sta ja tietokannasta, joiden avulla päivitetty versio onnenpyörästä kykenee hallitsemaan dataansa. Työhön sisältyi uusien ominaisuuksien kehittämistä, mutta myös vanhojen päivittämistä käyttämään tätä uutta backend-toteutusta.

Tavoitteet, jotka työllä oli, saavutettiin. Uudessa onnenpyörän versiossa on nyt backend, joka mahdollistaa datan tallentamisen, vaikka onnenpyöräsovellus pysäytettäisiinkin. API:n myötä onnenpyörän mahdollisuuksia on myös laajennettu. Vaikka käyttäjäkokemuksena onnenpyörän toiminnallisuus on pysynyt samana kirjautumista lukuun ottamatta, käyttäjät voivat nyt omistaa useampia listoja, joita onnenpyörä voisi käyttää arpomiseen. Onnenpyörää voisi esimerkiksi jatkokehittää siten, että käyttäjä voisi sovellusta käyttäessään vaihdella näiden listojen välillä ja lisäillä uusia listoja.

API:n avulla olisi myös helppoa automatisoida listojen luomista. Kun projekti laitetaan ensimmäistä kertaa pystyyn, niin se ei sisällä yhtään käyttäjiä eikä listoja. Tätä varten voisi esimerkiksi luoda skriptejä valmiiden listojen luomiseen käyttäjille. Käyttäjät voisivat myös mahdollisesti tuoda valmiita listoja esimerkiksi XML-tiedostona.

LÄHTEET

Nodejs. 2025. About Node.js. Luettu 5.4.2025. https://nodejs.org/en/about

Express. 2025. Express. Luettu 5.4.2025. https://expressjs.com/

Auth0. 2025. JSON Web Tokens. Luettu 5.4.2025. https://auth0.com/docs/secure/tokens/json-web-tokens

Mooc. 2025. Web-palvelinohjelmointi Java 2021. Luettu 5.4.2025 https://web-palvelinohjelmointi-21.mooc.fi/osa-6/4-rajapinnat-ja-rest

Internet Engineering Task Force. 2015. JSON Web Token (JWT) Luettu 5.4.2025 https://datatracker.ietf.org/doc/html/rfc7519

Pang, A. 2021. Crud Operations Explained. Luettu 5.4.2025 https://medium.com/geekculture/crud-operations-explained-2a44096e9c88

React.dev. 2025. Passing Data Deeply with Context. Luettu 5.4.2025 https://react.dev/learn/passing-data-deeply-with-context