1. Consistent use of Object-Oriented JavaScript principles

Tämä kriteeri edellyttää, että lopputyössä käytetään johdonmukaisesti objektorientoituneen ohjelmoinnin periaatteita. Objektorientoituneessa ohjelmoinnissa (OOP) keskitytään olion muodostamiseen, ja niiden toiminnallisuudet ja ominaisuudet ohjelmoivat saman yleisen "luokan" alle. JavaScriptissä voi tehdä tämän prototyypillisen perinnän tai ES6+:n luokkien avulla. Jotkin huomioon otettavat seikat ovat:

- **Luokat ja oliot:** Luomalla uudelleenkäytettäviä luokkia ja olioita.
    - JavaScriptin uudemmat versiot (ES6 ja siitä eteenpäin) tuovat mukanaan **class**-syntaksin, jota voi käyttää objektisuuntautuneen ohjelmoinnin periaatteiden, kuten perinnän, toteuttamiseen.
- **Perintä:** Mahdollista koodin uudelleenkäyttöä luomalla alaluokkia.
    - Käytä **extends**-avainsanaa laajentaaksesi toista luokkaa ja periyttääksesi sen ominaisuudet.
- **Kapselointi:** Rajaa pääsyä ja muokkausta luokan sisäisiin tietoihin ja metodeihin.
- **Polymorfismi:** Uudelleenmäärittele ja/tai laajenna metodi alaluokissa.
    - Määritä ylemmän luokan metodeja uudelleen alemmissa luokissa.

2. Advanced logic, looping through data, and dynamic DOM updates

Tämä kriteeri keskittyy ohjelmointilogiikan kehittyneempiin osa-alueisiin, mukaan lukien:

- **Ehtolausekkeet:** Käytä ehtoja, kuten if, else if, ja switch, hallitaksesi loogisia päätöksiä.
- **Datan iteroiminen:** Käytä erilaisia JavaScriptin loopeja (kuten for, for...of, forEach, map, jne.) käydäksesi läpi tietorakenteita.
- **Dynaamiset DOM-päivitykset:** Aktiivinen ja jatkuva interaktio käyttöliittymän kanssa käyttäen JavaScriptiä (esim. AJAX, Fetch API) päivittämään, lisäämään tai poistamaan DOM-elementtejä ja niiden sisältöä reaaliaikaisesti.
    - Käytä Document Object Model (DOM) APIa, kuten **getElementById**, **querySelector**, jne. päivittääksesi web-sivun sisältöä dynaamisesti.

3. Effective use of asynchronous data to enhance user experience

Asynkronisen datan tehokas käyttö vaatii seuraavia:

Asynkroniset kutsut: Käyttö API-kutsuissa tai muissa asynkronisissa toiminnoissa, esimerkiksi *fetch* tai *axios* kirjaston avulla.

***Promise*- TAI *async/await* -käsittely:** Datapyyntöjen hallintaan ja virheenkäsittelyyn. Käytä sitä tapaa, kumpi tuntuu tutummalta. Alla esimerkit molemmista:

Promise:

```
// Kapseloitu funktio
function fetchDataFromAPI(url) {
    return new Promise((resolve, reject) => {
        fetch(url)
            .then(response => {
                if (!response.ok) {
                    reject(`HTTP error! Status: ${response.status}`);
                } else {
                    resolve(response.json());
                }
            })
            .catch(error => {
                reject('Network error: ' + error.message);
            });
    });
}

// Funktion kutsu ja sen palauttaman Promisen käsittely
fetchDataFromAPI('https://api.example.com/data')
    .then(data => {
        console.log(data);  // Tulostaa haetun datan
    })
    .catch(error => {
        console.error('Tapahtui virhe:', error);  // Käsittelee mahdolliset
    });
```

Tässä esimerkissä `**fetchDataFromAPI**` on kapseloitu funktio, joka palauttaa `**Promise**`n. Tämä `**Promise**` pohjautuu sisäiseen `**fetch**`-kutsuun. Jos `**fetch**` epäonnistuu (verkkovirheen vuoksi) tai palauttaa ei-ok-statuskoodin (esim. 404 Not Found), funktio hylkää `**Promise**`n. Muussa tapauksessa funktio ratkaisee `**Promise**`n palauttamalla datan.

Voit sitten kutsua `**fetchDataFromAPI**`-funktiota ja käsitellä sen palauttamaa `**Promise**`a normaalilla `**.then()**` ja `**.catch()**` syntaksilla.

async/await:

```
// Kapseloitu funktio
async function fetchDataFromAPIAsync(url) {
    let response = await fetch(url);
    if (!response.ok) {
        throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return await response.json();
}

// Funktion kutsu ja sen palauttaman datan käsittely
async function handleData() {
    try {
        let data = await fetchDataFromAPIAsync('https://api.example.com/data
        console.log(data);  // Tulostaa haetun datan
    } catch (error) {
        console.error('Tapahtui virhe:', error);  // Käsittelee mahdolliset
    }
}

// Kutsutaan yllä määriteltyä handleData-funktiota
handleData();
```

Tässä esimerkissä on kaksi funktiota:

`fetchDataFromAPIAsync`: Tämä on kapseloitu asynkroninen funktio, joka tekee verkkopyynnön ja palauttaa datan tai heittää virheen.
`handleData`: Tämä funktio kutsuu ensimmäistä funktiota ja käsittelee sen palauttaman datan tai mahdolliset virheet.

`handleData`-funktiota voidaan sitten kutsua suorittamaan verkkopyyntö ja käsittelemään sen tulokset.

Käyttäjäkokemuksen parantaminen: Datan dynaamista päivittämistä käyttäen esimerkiksi ilmoituksia, latausindikaattoreita tai dynaamisia filttereitä ja lajitteluja, jotka parantavat käyttäjän vuorovaikutusta sovelluksen kanssa. Käytä JavaScriptiä suodattaaksesi ja lajitellaksesi tietoja reaaliaikaisesti käyttäjän syötteiden perusteella, parantaaksesi käyttökokemusta.