Dokumentation Slutprojekt

Inledning

Jag har utvecklat en anpassningsbar startsida för webbläsare med syfte att ge användare en överblick över deras dag. Sidan inkluderar widgetar för nyheter, temperatur, och en graf som visar temperaturen över de kommande 5 dagarna. Användaren kan skräddarsy sidan efter eget behov genom en inställningsmeny, där inställningarna sparas mellan olika sessioner.

Projektet

Utvärdering av din projektplanering

- **Vecka 42:** Projektplanering.
- **Vecka 43:** Skapande av statiska sidor med temporära värden för designöversikt.
- **Vecka 44:** Implementering av datahämtning med LocalStorage.
- **Vecka 45:** Visning av datan på sidan.
- **Vecka 46:** Utveckling av inställningssidan, spara ner inställningar och skapa en 'website builder'.
- **Vecka 47:** Läsning av inställningar från JSON och visa widgets enligt användarens preferenser.

Min projektplanering har fungerat bra och jag har kunnat hålla mig till den. Det som hade kunnat utvecklas är längden av projektet. Eftersom jag hade mer tid på mig efter v47 så har jag lagt ner mer tid på sidan som inte var planerad för i tidsplaneringen. Sen så har jag även behövt buggfixa och polera sidan, och det var inte heller med i planeringen. Det var då bra att jag inte fyllde ut alla veckor från början så att jag kunde ta det som de kom.

Testning

Jag har kört HTML koden via https://validator.w3.org/ och den har inte hittat några problem. det här kan delvis vara för att min kod har varit bra, men också delvis för att min HTML-kod laddas via javascript. Jag har däremot validerat sidan som ett firefox-inlägg, och då gav den varning på ‘unsafe practices'. Det var därför att jag hade onclick=”window.open()” i en div. jag fixade säkerhetsrisken med att lägga diven inuti en a tag.

Jag har testat sidan på firefox, chrome och edge, och alla webbläsare fungerar bra. Jag använder mig av backdrop-filter, som tidigare inte har varit accepterat av firefox, men nu så är det inte ett problem. Eftersom jag inte riktar mig till mobilanvändare så är det inte så viktigt att webbsidan ska fungera för dem, men jag har ändå anpassat sidan för mobil så att den fungerar korrekt.

Jag har även fått mina kompisar att testa på sidan, och de har gett mig feedback som jag har fixat. Några punkter var:

- news-delen fungerar inte
- dålig mobilanpassning
- dropdowns som overlappar konstigt
- enhet saknas på sidan

Det 3 senaste punkerna var enkla att fixa med css, men en svårare del var att news-api inte gav några nyheter. Det här beror på att kontot jag har på newsapi är ett gratiskonto, och hämtning av data endast kan hända från localhost. Ett betalkonto kostar tusentals i månaden, så det var inte en lösning. Lösningen är istället att datan körs lokalt på en server med flask, som sedan läggs ut på en sida i plaintext.

Tekniska Utmaningar och Lösningar

- Hantering av asynkrona API-anrop och komplex data.
- Skapande av ett responsivt och dynamiskt gränssnitt.
- Implementering av LocalStorage för effektiv datalagring och hämtning.

En stor utmaning med mitt projekt har varit organiseringen av min kod. Webbsidan består utav många rörliga delar, och pga det så har nästan största delen av min tid gått ut på att komma på effektiva och fungerande lösningar för olika delar av koden.

Upphovsrätt och GDPR

- Användningen av bilder och innehåll hanterades genom att endast använda royaltyfria resurser eller sådant som skapats in-house.
- API:er valdes på grundval av deras överensstämmelse med GDPR.

Med openweathermap så får man använda API:n för personligt bruk. Alla ikoner har tagits från bootstrap.

Kodanalys

graph.js

upd_dates()

- upd_dates() består av 2 delar med vardera huvud-loop
- del 1 går igenom alla element som har klassen “dates” och lägger till nuvarande dag + 4 dagar framåt + en “all” dag.
- vi går igenom alla dates element i en forloop för att användare kan välja att ha fler än 1 graf. på dehär sättet motverkar vi både error av för få och för många grafer. Den här tekniken uppdateras på flera ställen i min kod.
- del 2 av funktionen lägger till eventlisteners för varje dag så att användare kan klicka på dagar för att uppdatera grafen.

```
const index = Array.from(e.target.parentElement.children).indexOf(e.target)
```

Betyder att vi kollar på elementet som användaren klickar på, och sen så kollar vi på dens “parent”. Vi kollar sen på alla föräldrerns barn, och vi hittar index på vårat som användaren har klickat på. Vi adderar sen index med ett nytt date objekt för att filtrera bort temperaturer som inte har vårat datum anknutet med sig.

```
const all_temp = data.list.map(d => d.main.temp)
const min = Math.min(...all_temp) - 1
const max = Math.max(...all_temp) + 1
```

Vi använder oss av “...” på flera olika ställen i min kod. Det här är en “förkortning” som i det här tillfället betyder att vi delar upp en array i alla sina bitar. En längre version av samma kod skulle se ut såhär:

```
let smallest_number = 999
for (let i = 0; i < all_temp; i++) {
    if (i < smallest_number) {
        i = smallest_number
    }
}
```

som man ser så är “...” mycket smidigare.

upd_graph()

- Vi callar upd_graph() från upd_dates, och vi matar in argumentet list, min och max.
- som i dates funktionen så kollar vi efter alla canvas objekt på sidan, eftersom användare kan skapa flera grafer.

custom events

För att rita grafen så behöver vi ha något att rita. Eftersom vi inte har väderdatan lokalt så behöver vi få den från internet, och med det så finns det lite delay som vi behöver räkna med. Vi laddar datan från ett annat dokument för att ha bättre kodstruktur, och för att säga till graph.js att vi nu har datan som ska laddas så skapar vi ett custom event som vi lyssnar för i graph.js.

```
document.addEventListener('weather_data_loaded', () => {...})
```

Koden är precis som om vi skulle vänta på DOMContentLoaded, ända skillnaden är att vi lyssnar efter vårt egna event.

colorpicker.js

Jag valde att skriva min egna colorpicker istället för att använda någon annans. Det här var delvis för att det var svårt att hitta simpla color pickers som var free-use, och delvis för att jag tyckte att det skulle vara en rolig challenge.

colorpickern är skriven så modulärt som möjligt, så att jag kan återanvända koden i andra projekt eller dela ut den till andra. Såhär använder du colorpickern:

- skapa en div med en ‘.pick’ klass
- importera get_colors i din javascript-fil.

update_canvas()

För att vi har 120200 gånger som vi går igenom loopar för att rita varenda pixel i colorpickern så tar det lite tid att ladda den. För att dra ner på tiden så börjar den köras så fort sidan har laddats, och sen togglar vi bara visibility på den när vi behöver den. update_canvas() funktionen körs också konstant.

```
for (let i = 0; i < 200; i++) {
    const hue = i / 200
    const color = `hsl(${hue * 360}, 100%, 50%)`
    ctx.fillStyle = color
    ctx.fillRect(210, i, 25, 1)
}
```

Colorpickern består utav 2 gradients, och den här koden ritar den högra delen. vi använder hsl för att det är en lätt uträkning som inte tar så lång tid, då vi bara räknar ut hue.

```
for (let i = 0; i < 200; i++) {
        for (let j = 0; j < 200; j++) {
            let rgb = [0, 0, 0]
            for (let k = 0; k < 3; k++) {
(i/200)) - j/200*255[k] = Math.round(255 * (1 - (i/200)) + max_color[k] *
            }
            ctx.fillStyle = `rgb(${rgb[0]},${rgb[1]},${rgb[2]})`
            ctx.fillRect(i, j, 1, 1)
        }
    }
```

1. for (let i = 0; i < 200; i++) {: En yttre loop som itererar genom varje pixel i x-riktning (horisontellt) från 0 till 199.
2. for (let j = 0; j < 200; j++) {: En inre loop som itererar genom varje pixel i y-riktning (vertikalt) från 0 till 199.
3. let rgb = [0, 0, 0]: Skapar en array rgb som kommer att hålla de tre färgkomponenterna (röd, grön och blå) för varje pixel.
4. for (let k = 0; k < 3; k++) {: En inre loop som itererar genom varje färgkomponent (r, g, b).
5. rgb[k] = Math.round(255 * (1 - (i/200)) + max_color[k] * (i/200)) - j/200*255: Beräknar färgvärdet för varje färgkomponent baserat på pixelns position. Detta är där färggradienten skapas. max_color är en variabel som innehåller de maximala färgvärdena för varje färgkomponent.
6. ctx.fillStyle = rgb(${rgb[0]},${rgb[1]},${rgb[2]})``: Ställer in fyllningsfärgen för Canvas-contexten (ctx) till den beräknade färgen.
7. ctx.fillRect(i, j, 1, 1): Fyller en enskild pixel på position (i, j) med den aktuella färgen. Eftersom varje pixel är bara 1x1 stor, skapas ett “mosaikmönster” av färgade pixlar på Canvas.

```
document.addEventListener('mousedown', () => m.down = true)

document.addEventListener('mouseup', () => m.down = false)
```

Den här koden används för att kolla state på mustryck, som jag sedan använder mig av i mousemove eventlisterna. I min mousemove så kollar jag varg musen är i relation med colorpickern, och sen så får jag pixeln som musen är över. Med den informationen så använder jag koordinaterna till ett getImageDatta anrop. Det är en inbyggd funktion som läser pixelns värde. Den här koden skulle kunna optimeras med att spara alla färger i en matrix och sen ta den relativa muspositionen som värden i matrisen. Jag tror att koden skulle bli snabbare då, men det är ganska mycket onödigt jobb då min lösning funkar bra nu.

load_html.js

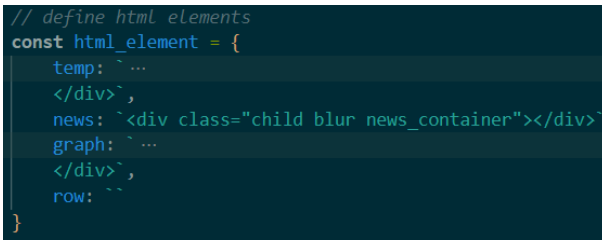
load_html()

```
    let page_content = localStorage.getItem('widget_names')
    if (!page_content) {
        localStorage.setItem('widget_names', ['row', 'graph'])
        load_html()
    }

    let row_content = localStorage.getItem('row_content')
    if (!row_content) {
        localStorage.setItem('row_content', ['temp', 'news'])
        load_html()
    }

    page_content = page_content.split(',')
    row_content = row_content.split(',')
```

Vi försöker få info på vad som ska finnas på sidan med den här koden. Om vi inte har satt widget_names tidigare så sätter vi den här och kör om funktionen. Det här är ett bra sätt att hantera default values, men kan orsaka problem på settings sidan om användare inte har laddad in huvudsidan förut.



html_element är en klass som definerar html koden som ska användas. Row är tom eftersom vi sätter den på nästa rad.

```
html_element.row = `
        <div class="col">
            ${html_element[row_content[0]]}
        </div>
        <div class="col">
            ${html_element[row_content[1]]}
        </div>`
```

Vi sätter content i row med värden från htm_element klassen.

```
page_content.forEach(html => {
        const div = document.createElement('div')
        div.className = 'c'
        div.innerHTML = html_element[html]
        document.body.appendChild(div)
    })
```

Vi loopar igenom vår lista med element som ska finnas med på sidan, och sätter ut dem. Efter den här koden körs så callar vi apply_settings() som sätter all styling.

På samma sätt som vi kollar om värde finns och sätter default värde på page och row content så gör vi samma för ‘bgtype’. Default är ‘color’.

I color så tar vi vårt sparade / default värde och manipulerar det till flera children i en lista.

```
            // set bg color
            document.documentElement.style.background = bg_colors[0]
            // set gradient
            if (bg_colors.length == 1) break
            let gradient = 'linear-gradient(to bottom, '
            for (let i = 0; i < bg_colors.length; i++) {
                gradient += bg_colors[i]
                if (i != (bg_colors.length - 1)) {
                    gradient += ', '
                }
            }
            gradient += ')'
            document.body.style.background = gradient
```

Här tillämpar vi den sparade färgen till sidan. Ifall vi bara har en färg så behöver vi inte tänka på gradient, men om vi har fler så har vi de. Vi loopar då igenom alla färger och lägger till dem i en string, med ett kommatecken för alla children utom den sista.

update_clock()

Funktionen körs varje sekund och sätter rätt tid på klock-element. Vi börjar med att skapa ett nytt date obj och definera timmar, minuter, sekunder, dagar, månader, år. De här är integers, vilket betyder att de bara är siffror. t.ex så blir 9 bara nio, men vi vill inte displaya 9:9:9 som tid för att det är fult. För att fixa det har vi en funktion som heter format_num(). Där kollar vi om värdet är < 10, och lägger till en nolla ifall den är de.

```
// grab all clock div elements
const clocks = document.querySelectorAll(".clock")
// loop thru all clocks
clocks.forEach(clock => {
    // update clock
```

```
    clock.innerHTML = formatted_time
})

// same thing for date
const dates = document.querySelectorAll(".date")
dates.forEach(date => {
    date.innerHTML = formatted_date
})
```

Igen, eftersom användare kan skapa mer än 1 clock och date element så måste vi loopa igenom alla på sidan och sätta innerText till formatted_time. window.setTimeout andvänds för att calla funktionen varje sekund.

fetch_data.js

- Användning av geolokalisering för att hämta användarens position.
- Fetching av väderdata från OpenWeatherMap-API.
- Hantering av säkerhet och felmeddelanden vid hämtning av geolokalisering och nyhetsdata.

Använda Resurser

Jag tog 2 delar av min kod i colorpicker.js. Det här var 2 färg konverterar-funktioner, en rgb till hex och en hsv till rgb. Anledningen till att jag tog den här koden är att det finns väldigt få sätt att skriva de här funktionerna, och att det bara är ren matte (som jag inte är jättebra på). Jag har länkat till sidorna jag tog funktionerna ifrån i kode.

Säkerhet

Jag har skrivit all kod själv och allt är lokalt kört. Inga användaruppgifter används. Den enda koden som är skriven av någon annan är 2 matte funktioner i colorpicker.js. Där har jag modifierat koden lite, och jag förstår den helt.

Betyg

Jag tycker att mitt projekt förtjänar ett A. Jag har använt mig av många olika tekniker, som custom events, api-requests och localstorage. Jag har också väldigt mycket kod som körs i samspel. Webbsidan är också polerad och användbar. Modulariteten av koden tycker jag också förtjänar pluspoäng.