GGisEZ

1 Formålet med applikasjonen

Formålet med applikasjonen er å introdusere arbeidsgangen i et typisk geografisk informasjonssystem (GIS) på en enklest mulig måte. Det er derfor applikasjonen har fått det noe kryptiske navnet "GGisEZ" - en kombinasjon av forkortelsen "ggez" (good game - easy) fra videospillverdenen og forkortelsen GIS. Navnet har det formål å hentyde til at GIS ikke trenger å være vanskelig - at GIS kan være "ez". Framtiden vil vise om intensjonen med navnet i det hele tatt kommer fram hos den gjennomsnittlige brukeren av applikasjonen.

Det viktigste fokusområdet under utviklingsprosessen var brukervennlighet. Ettersom applikasjonen er myntet på personer med lite erfaring med GIS, gikk en stor porsjon av arbeidet med til å "idiotsikre" bruker-input så godt det lar seg gjøre under de gitte tidsrammene. Derfor har hvert av de ulike GIS-verktøyene en egen valideringsfunksjon som sjekker om de kartlagene brukeren ønsker å utføre transformasjonen gir mening for det valgte verktøyet. Eksempelvis skal det være umulig å "intersection" på et kartlag bestående av linje-geometrier og punkt-geometrier, da dette er en transformasjon myntet på polygon-geometrier. Ønsket om brukervennlighet reflekteres også i valget av programmingsspråk og i brukergrensesnittet.

2 Bakgrunn for valg av programmeringsspråk

2.1 TypeScript

Programmingsspråket jeg valgte å bruke for dette prosjektet, er TypeScript. TypeScript er et superset av JavaScript som innfører et typesystem til det ellers dynamiske språket. TypeScript er først og fremst ment som et verktøy for utvikleren, og transpileres til JavaScript før det sendes til nettleseren. Fordelen med å bruke TypeScript er at du varsles når du er i ferd med å gjøre uriktige ting, slik som å sende ugyldig input til en funksjon. Det gjør utvikler i stand til å oppdage feil i "compile time", framfor i "runtime" som man måtte gjort dersom man brukte JavaScript. Dette kan spare deg masse tid som ellers hadde vært brukt på feilsøking og brannslukking. TypeScript gjør også at du lett kan finne ut av hvilke funksjoner om finnes på et gitt object, uten å måtte lete deg fram til filen hvor klassen for det objektet er definert.

TypeScript kan være nyttig når man jobber med GeoJSON. Jeg har laget en rekke funksjoner som sjekker om et gitt objekt er gyldig GeoJSON og om det er en Feature, FeatureCollection, GeometryCollection, Point, LineString, MultiPolygon, osv. Disse funksjonene fungerer som "type guards":

```
function foo(data: GeoJSON) {
    if (isFeatureCollection(data)) {
        // Vi vet nå at "data" er en FeatureCollection og har nå tilgang til
        // "features"-attributten som enhver FeatureCollection er pålagt å ha
        ...
    }
}
```

"Type guards" hjelper utvikleren med å skrive trygg kode hvor man unngår å bruke attributter og funksjoner som ikke er definert på objektet man jobber med.

2.2 SvelteKit

For å gjøre utviklingsprosessen enklere, har jeg valgt å benytte meg av et app-rammeverk ved navn SvelteKit. SvelteKit er bygget på komponent-rammeverket Svelte, som er et nytt og veldig populært valg for de som ønsker å bygge nettsider. Svelte/SvelteKit kjennetegnes ved å være raskt i produksjon og enkelt å bygge apper med. Sistnevnte er hovedgrunnen til at jeg valgte nettopp dette rammeverket, da noe som ReactJS ofte gjør at man må skrivde det som virker som unødvendige mengder "boilerplate"-kode. Jeg ville også bare prøve noe nytt, og SvelteKit virket som et godt valg, ettersom det virkelig er i vinden i frontend-verden.

3 Oppsummering av arbeidsgangen

4 Programstruktur

```
src
|
└──lib
|   └──components
|   |   └──AnalysisTools
|   |       |   file111.txt
|   |       |   file112.txt
|   |       |   ...
|   |   └──GeoJsonProcessing
|   |       |   bbox.ts
|   |       |   bboxClip.ts
|   |       |   buffer.ts
|   |       |   ...
|   |   |   ...
|   |   └──Map
|   |       |   ...
|   |   └──Sidebar
|   |       └──ToolOptions
|   |       |   ListItem.svelte
|   |       |   Sidebar.svelte
|   |   |   ...
|   └──scss
|       |   ...
|   └──utils
|       fileUploader.ts
|       geojson.ts
└──routes
|   |   _global.scss
|   |   +page.svelte
|   |   ...
└──routes
|   |   mapLayers.ts
|   |   mapSources.ts
└──theme
|   |   ...
|   |   _smui-theme.scss_
|   ...
|   app.html
```

```
└──static
|   |   ...
|   |   smui.css
|
```

Over er den overornede strukturen til programmet, hvor mindre viktige filer er ekskludert. `app.html` er "startpunktet" til applikasjonen, og den linker videre til Svelte-delen av applikasjonen. I `routes/`-mappen finner man alle endepunktene, og fil- og mappestrukturen her bestemmer URL-ene til de ulike sidene i applikasjonen. Her har vi kun én `+page.svelte` - altså har vi kun én side i denne applikasjonen.

SvelteKit gir deg også en `lib/`-mappe der du kan legge inn alt fra enkeltkomponenter til CSS-filer. Det er anbefalt å ha mest mulig kode her, slik at ikke filene i `routes`-mappen blir for store, da disse bør fokusere på den overordnede strukturen til applikasjonen, og ikke implementasjon.

I `lib/`-mappen har jeg mapper for hvert av de tre panelene i applikasjonen (`AnalysisTools`, `LayerInfo` og `Sidebar`), samt en egen for selv `map`-komponenten. `Map`-komponenten importer disse tre hovedkomponentene og plasserer dem i et overlay som lever oppå kartet. `Map`-komponenten importeres deretter til `+page.svelte`, slik at den vises når man navigerer til hjemmesiden (https://oskarhlm.github.io/GGisEZ/).

Videre har jeg en mappe kalt `GeoJsonProcessing/`. Er har jeg filer som `buffer.ts`, `intersect.ts` og `difference.ts`. Disse filene eksporterer et objekt bestående av en *processor* og en *validator*. Her er et eksempel fra `bboxClip.ts`:

```
export default {
    processor: bboxClipProcessor,
    validator: bboxClipInputValidator
} satisfies GeoJSONProcessor<
    MapLayer<mapboxgl.Layer>[],
    GeoJSON[],
    BBoxClipOptions,
    BBoxClipOptions
>;
```

Her eksporteres det et objekt av typen `GeoJSONProcessor` som tar inn en liste med `MapLayer`s og returnerer en liste med `GeoJSON`-objekter. De to `BBoxClipOptions`-ene definerer formen på `options`-objektene som sendes til henholdsvis *processor*-en og *validator*-en. Alle filene i denne mappen returnerer objekter av denne typen, noe som gjør at man ikke trenger å hardkode funksjonalitet når man skal bruke verktøyene i .svelte-filene.

Flere av verktøyene har komplementære .svelte-komponenter med navn som `BboxClipOptions.svelte` og `BufferOptions.svelte`. Dette er komponenter som settes inn over listen over kartlag i `Sidebar`-en, og er ansvarlige for at man skal kunne sende `options` til filene som ble diskutert i forrige avsnitt. Disse filene ligger under `Sidebar/ToolOptions/`.

Avslutningsvis vil jeg diskutere de ulike stedene man finner .scss-filer. SCSS er en preprosessor for CSS som skal gjøre CSS enklere å skrive ved hjelp av "nesting", mixin's, osv. Filer som definerer mixin's (CSS-funksjoner)

og variabler (fargepalett) ligger i `lib/scss/`. I tillegg har jeg en fil `_smui-theme.scss` som kompileres til `smui.css`. Denne bestemmer stylingen til alle Material UI komponenter som brukes applikasjonen. Dette sammenfatter alle knapper, sliders, osv.

5 Tutorial

6 Diskusjon

6.1 Problem underveis

6.1.1 TypeScript - a blessing and a curse

Selv om TypeScript kan gjøre programmet ditt både enklere å utvikle, mer forutsigbart og gi det større grad av korrekthet, kan det også by på utfordringer man ikke får ved bruk av et dynamisk programmeringsspråk. Det var utfordrende å ta hensyn til alle ulike former for GeoJSON som brukeren kan legge inn i kartet. Det kan være en form for Geometry (Point, LineString, Polygon, Multi-"x", GeometryCollection) og Feature's, FeatureCollection's. Det er spesielt viktig å vite hva slags geometrier et gitt kartlag består av når man skal utføre transformasjoner på dem. Funksjonene i TurfJS - JavaScript-biblioteket jeg har brukt for å utføre disse transformasjonene - er avhengige av at man gir dem data av riktig format. Noen funksjoner ønsker kun Geometry- og Feature-objekter, og tillatter ikke at du gir dem GeometryCollection's og FeatureCollection's. Selv om dette ikke støttes direkte, er det ønskelig at man skal kunne utføre transformasjonene der det er mulig. Derfor

6.2 Mangler og feil

6.3 Brukergrensesnitt

Som nevnt i innledningen, var brukervennlighet hovedfokuset i utviklingsprosessen. I tillegg til validering av bruker-input, er også brukergrensesnitt helt avgjørende her. Under utformingen av brukergrensesnittet har jeg fokusert på at det skal være færrest mulig trykk mellom et ønske om å utføre en analyse til analysen er utført og et nytt kartlag er opprettet. Det bør også være flere intuitive måter å utføre samme funksjon på, slik at hver bruker kan bruke applikasjonen på den måten han eller hun vil.

6.3.1 Lagoversikt

Ytterst til venstre i applikasjonen finner man et panel med oversikt over alle kartlag som ligger inne i kartet. Plassingen av denne samsvarer med den man finner i de fleste populære GIS-applikasjoner (se ArcGIS, QGIS, etc.). I listen over kartlag kan man dra enkeltlag opp og ned og sortere lagene slik man vil. Kartlagene øverst i listen vises foran kartlag lenger ned i listen, slik man forventer. Hvert listeelement har også knapper for å endre synlighet og sletting, samt et ikon helt til venstre for å vise om kartlaget består av punkter, linjer eller polygon.

Lagoversikten i sin "default"-modus er ment for å håndtere alt av synligheten til kartlagene, og ingenting mer. Det er ønskelig å styre synlighet og sletting av et enkelt kartlag ved kun ett trykk, heller enn at man eksempelvis må høyreklikke og skumlese seg nedover en liste av operasjoner helt til man finner "Delete layer". I en mer avansert applikasjon er dette helt klart den beste løsningen, men for en applikasjon med fokus på brukervennlighet mener jeg at det er bedre å ha disse knappene på øverste synlighetsnivå. Jeg ønsket også å unngå så mange "popup"-bokser og "modals" som mulig, alt for å gjøre applikasjonen enklere og mer oversiktlig.

6.3.2 Toolbar

Til høyre for lagoversikten finner man en kolonne med oversikt over tilgjengelige GIS-verktøy. Kun ett verktøy kan velges av gangen og når et verktøy er valgt kan man bruker lagoversikten til å velge hvilke(t) lag man ønsker å utføre en transformasjon på. "Checkboxes" og en uthevingsfarge brukes for å tydelig vise hvilke lag som er valgt. Ved å gjenbruke lagoversikten til å velge kartlag unngår man å måtte ha et eget vindu når man skal utføre en transformasjon, noe jeg mener forbedre brukervennligheten. Når man har valgt lag som er gyldige for den valgte transformasjonen, får man muligheten til å trykke på "Apply"-knappen som nå har blitt blå og trykkbar.

Hvis jeg skal være litt selvkritisk her, vil jeg si at ikonene for å illustrere funksjonaliteten til de ulike ikonene kan være litt vanskelige å tyde. Det er en kunst å gjøre små ikoner beskrivende nok, og det er mulig at jeg burde ha valgt enklere (mer rektangulære) former. Ikonene for "union" og "dissolve" kunne kanskje til fordel være mer ulike, samtidig som det må påpekes at dette er nært beslektede transformasjoner. Mens ikonene kanskje kunne være mer beskrivende, synes jeg likevel at de tar seg godt ut rent estetisk, og liker at de matcher med farge-paletten som er valgt for applikasjonen som en helhet.

6.3.3 Styling

Hvis man trykker på et listeelement/kartlag i lagoversikten dukker det opp et panel til høyre der man får muligheten til å endre navn og farge på det valgte kartlaget. Det nåværende navnet og fargen vil være startverdiene når man åpner dette panelet. Dette panelet er kun synlig dersom man faktisk trykker på et kartlag, og er der ikke når man først starter applikasjonen. Dermed unngår man "information overload", noe som er vanlig å få når man åpner et GIS-program for første gang. For å forbedre brukervennligheten har jeg prøvd å ha minimalt med funksjonalitet synlig ved start, slik at brukeren tvinges til å fokusere på det som er viktig, først.

Styling-panelt kan lukkes ved enten å trykke på kartlaget på nytt, eller å trykke på "x"-symbolet øverst til høyre i panelet, igjen for å ha flere intuitive måter å gjøre samme ting på.

6.4 Potensielle forbedringer

Referanser