Table of Contents

GeoStyler - Workshop

Herzlich Willkommen beim **GeoStyler - Workshop**. Dieser Workshop soll Ihnen einen einführenden Überblick über den GeoStyler als ein webbasiertes Werkzeug zur interaktiven Erstellung von kartographischen Style-Vorschriften für Geodaten geben.

Grundsätzlich werden wir in diesem Workshop eine einfache Anwendung schreiben, mit der Sie eine Story Map über CoVid-19 erstellen können. Die fertige Anwendung, wie Sie sie in diesem Workshop "bauen" werden, kann hier eingesehen werden.

> Wenn Sie diese Seite auf Ihrem eigenen Gerät besuchen oder die PDF-Version ausdrucken möchten, dann können Sie die Workshop-Materialien hier herunterladen.

Setup

Die folgenden Anweisungen und Übungen setzen voraus, dass Sie bestimmte Anforderungen an ihre lokale Maschine erfüllt haben. Bitte prüfen Sie, ob Sie folgende Voraussetzungen installiert haben:

- Einen geeigneten Text Editor, wie z.B.Atom.
- NodeJS in Version 6.

> Die aktuell auf ihrer lokalen Maschine installierte Node Version können Sie mit dem Befehl `node -v` in Ihrem Terminal ansehen. Falls diese Version nicht der für diesen Workshop notwendigen Version 6 entspricht, können Sie ganz einfach mit dem Befehl `nvm use 6` die Node Version wechseln.

Alles eingerichtet? Dann los geht's!

Überblick

Der Workshop ist aus einer Reihe von Modulen zusammengestellt. In jedem Modul werden Sie eine Reihe von Aufgaben lösen, um ein bestimmtes Ziel zu erreichen. Jedes Modul baut Ihre Wissensbasis iterativ auf.

Die folgenden Module werden in diesem Workshop behandelt:

- Grundlagen - Tauchen Sie in die Grundlagen von EcmaScript 6, React und npm ein.
- Erste Schritte - Lernen Sie, wie Sie Ihre eigene React-App erstellen und wie Sie react-geo darin einbinden können.
- GeoStyler einbinden - Erweitern Sie ihre Anwendung mit GeoStyler.

Autoren

- Jan Suleiman (suleiman@terrestris.de)
- Dirk Mennecke (-)

Basics

Bevor wir mit GeoStyler anfangen, werfen wir einen kurzen Blick auf einige grundlegende Voraussetzungen:

- NPM - Node / Node package manager / NVM
- ES6 - EcmaScript 6

Da es sich bei dem GeoStyler um eine React basierte Komponenten Bibliothek handelt, ist es unerlässlich, einen kurzen Überblick über React zu verschaffen.

- React - ReactJS

npm

npm ist der Paketmanager für Node.js (eine JavaScript-Laufzeitumgebung) und die weltweit größte Software-Registry (mehr als 600k Pakete) mit etwa 3 Milliarden Downloads pro Woche

Sie können npm verwenden, um:

- Pakete an Ihre Anwendungen anzupassen oder sie so einbinden, wie sie sind.
- Eigenständige Tools herunterzuladen, die Sie sofort verwenden können.
- Pakete ohne Herunterladen mit npx auszuführen.
- Code mit jedem npm-Benutzer überall zu teilen.
- Den Code auf bestimmte Entwickler zu beschränken.
- Virtuelle Teams (Orgs) zu bilden.
- Mehrere Versionen von Code und Code-Abhängigkeiten (dependencies) zu verwalten.
- Anwendungen zu aktualisieren, wenn der zugrunde liegende Code aktualisiert wird.
- Andere Entwickler zu finden, die an ähnlichen Problemen arbeiten.

package.json

Der Befehl `npm init` in Ihrem Projektordner öffnet einen interaktiven Dialog zur Einrichtung eines npm-Projekts. Das Ergebnis ist das `package.json` mit allen wichtigen Einstellungen, Skripten und Abhängigkeiten (dependencies) Ihres Projekts.

```
{
  "name": "name_of_your_package",
  "version": "1.0.0",
  "description": "This is just a test",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "http://github.com/yourname/name_of_your_package.
  },
  "author": "your_name",
  "license": "ISC"
}
```

Für weitere Informationen sei hiermit auf die npm docs hingewiesen.

Pakete mit npm installieren

Die gebräuchlichste Art, neue Pakete mit npm zu installieren, ist über die Kommandozeile. Um ein Paket zu installieren, geben Sie einfach folgendes ein:

```
npm install packagename
```

Sie finden die installierten Pakete im Unterordner `node_modules`.

Node version manager NVM

- bash-Skript zur Verwaltung mehrerer aktiver node.js-Versionen
- Siehe hier

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v
nvm i v8
```

ES6

ES (ECMAScript) ist eine markenrechtlich geschütze Spezifikation der Skriptsprache, die zur Standardisierung von JavaScript erstellt wurde. Wie der Name schon verrät, ist ES6 (später in ES2015 unbenannt) die sechste Ausgabe und wurde mit einer bedeutenden neuen Syntax zum Schreiben komplexer Anwendungen ausgestattet, einschließlich Klassen und Module. Einige Browser unterstützen ES6 nicht (oder nur teilweise), weswegen der ES6 Code in ES5 transiliert werden kann. ES5 genießt eine breitere Kompabilität.

JavaScript-Frameworks und Bilbiotheken zur Erstellung moderner Webanwendunge sind in ES6 geschrieben.

`import`

```
import { CircleMenu } from "react-geo";
```

`export`

```
const name = "Peter";
export default name;
```

Variablen Deklaration

- ES5: `var`
- ES6: `var` , `let` und `const` :
  - scope abhängig

Funktionen

```
// ES5
var myFunc = function(myArg) {
  if (!myArg) {
    myArg = "Peter";
  }
  return myArg + " is the best arg!";
};
// ES6
const myFunc = (myArg = "Peter") => {
  return myArg + " is the best arg!";
}; // myFunc() ----> 'Peter is the best arg!'
// ES6 shortened
const myFunc = myArg => myArg + " is the best arg!";
```

Template string

```
// ES5
var a = 1909;
console.log("Year: " + a);
// ES6
console.log(`Year ${a}`);
```

Destructuring assignment

Siehe auch hier.

Beispiel 1: Objekt Destrukturierung (Object destructuring)

```
// ES5
var obj = {
  name: "Peter",
  age: 55
};
var age = obj.age;

// ES6
const obj = {
  name: "Peter",
  age: 55
};
const { age } = obj;
```

Beispiel 2 (verwendet auch den Spread operator):

```
// ES5
var user = { name: "peter", age: 12 };
user = Object.assign(user, { email: "peter@love.de" });
// ES6
let user = { name: "peter", age: 12 };
user = { ...user, email: "peter@love.de" };
```

React

React ist eine moderne und quelloffene JavaScript-Bibliothek zur Erstellung von Benutzeroberflächen auf der Basis von ES6. Ursprünglich wurde react von einem Software-Ingenieur bei Facebook entwickelt und wird weiterhin von Facebook (unter anderem) gewartet.

Mit React können Entwickler Webanwendungen erstellen, die Daten verwenden, welche sich im Laufe der Zeit ändern können, ohne die Seite neu zu laden. Es zielt in erster Linie auf Geschwindigkeit, Einfachheit und Skalierbarkeit ab. React verarbeitet nur Benutzeroberflächen in Anwendungen. Dies entspricht dem View im Model-View-Controller (MVC)-Muster und kann in Kombination mit anderen JavaScript-Bibliotheken oder Frameworks in MVC, wie z.B. AngularJS, verwendet werden.

Das kleinste React-Beispiel sieht wie folgt aus:

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);>
```

Folgen Sie den docs und dem Tutorial für weitere Informationen.

Props

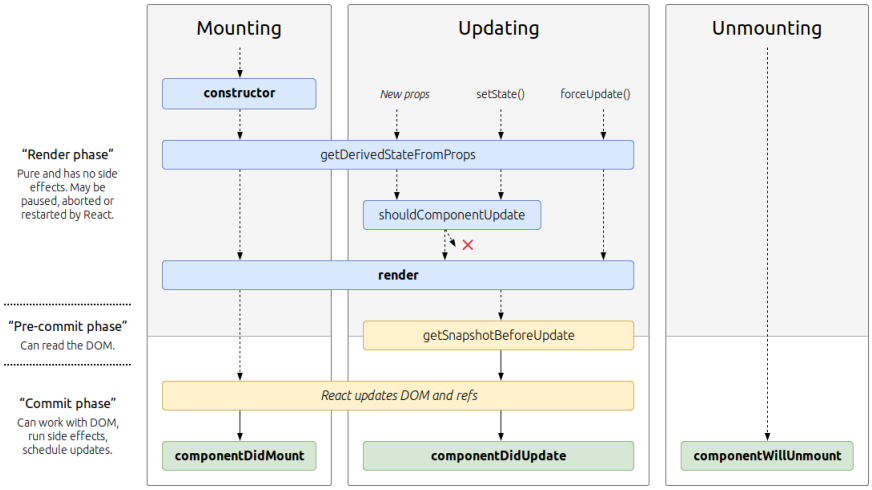
Props sind die Konfigurationen der Komponente, die Sie an Instanzen übergeben. Sie werden von der oben genannten Komponente empfangen und sind unveränderlich. Ausführliche Informationen finden Sie unter Components und Props.

State

Die state speichert intere Werte einer Komponente. Es handelt sich hier um eine *serialisierbare* Darstellung eines Zeitpunkts - eine Momentaufnahme. Der Zustand kann innerhalb einer Komponente

über `setState` manipuliert werden. Ausführliche Informationen finden Sie unter State und Lifecycle.

Lifecycle



Bildquelle, letzter Zugriff 08.04.2020.

Siehe State und Lifecycle

JSX

React-Komponenten werden normalerweise in JSX geschrieben, einer JavaScript-Erweiterungssyntax, die das Zitieren von HTML und die Verwendung der HTML-Tag-Syntax zur Darstellung von Unterkomponenten ermöglicht. Die HTML-Syntax wird in JavaScript-Aufrufe des React-Frameworks verarbeitet. Entwickler können auch in reinem JavaScript schreiben. Ein Beispiel für JSX-Code:

```
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <div>
        <p>Header</p>
        <p>Content</p>
        <p>Footer</p>
      </div>
    );
  }
}

export default App;
```

In diesem Workshop werden wir Functions verwenden.

Erste Schritte

Nachdem wir die Grundlagen thematisiert und einen Einblick in npm, ECMAScript 6 und React erhalten haben, wird in diesem Kapitel eine react-basierte Webanwendung mit create-react-app erstellt.

Dieses Kapitel ist in folgende Subkapitel unterteilt:

React Applikation erstellen

Natürlich könnten wir diesen Workshop damit beginnen, dass wir eine auf React basierende Webanwendung von Hand erstellen, aber wie Sie sich vorstellen können, wäre dies eine schwierige Aufgabe für den Anfang. Wir wollen also direkt in React eintauchen, ohne die Notwendigkeit, alle Entwicklungswerkzeuge zusammenzuhalten, um eine Webanwendung zum Laufen zu bringen. Glücklicherweise gibt es ein Befehl, mit dem wir eine Anwendung für uns generieren können (auch ohne jegliche Konfiguration!): create-react-app.

Die Erstellung einer neuen Anwendung ist einfach. Hierfür muss nur zu einem Ordner Ihrer Wahl navigiert werden und dort mit dem Namen *geostyler-app* eine Anwendung innerhalb dieses Verzeichnisses erstellt werden:

```
npx create-react-app geostyler-app
```

Dies wird eine Weile dauern. Sobald die Erstellung abgeschlossen ist, kann mit folgendem Befehl in den Ordner des Projektes navigiert werden:

```
cd geostyler-app
```

Nun kann endlich der Entwicklungsserver gestartet werden. Hierfür muss folgender Befehl in das Terminal eingegeben werden:

```
npm start
```

Aufgabe 1. Erstellen Sie eine React Anwendung mit

```
npx create-react-app geostyler-app
```

Aufgabe 2. Navigieren Sie mit Ihrem Terminal in den Ordner und führen Sie dort den Befehl `npm start` aus.

> Dies wird nur funktionieren, wenn Sie node (Version 6), sowie npm und nvm erfolgreich auf ihrer lokalen Maschine installiert haben. Der Befehl, um node Version 6 zu aktivieren, lautet `nvm use 6`.

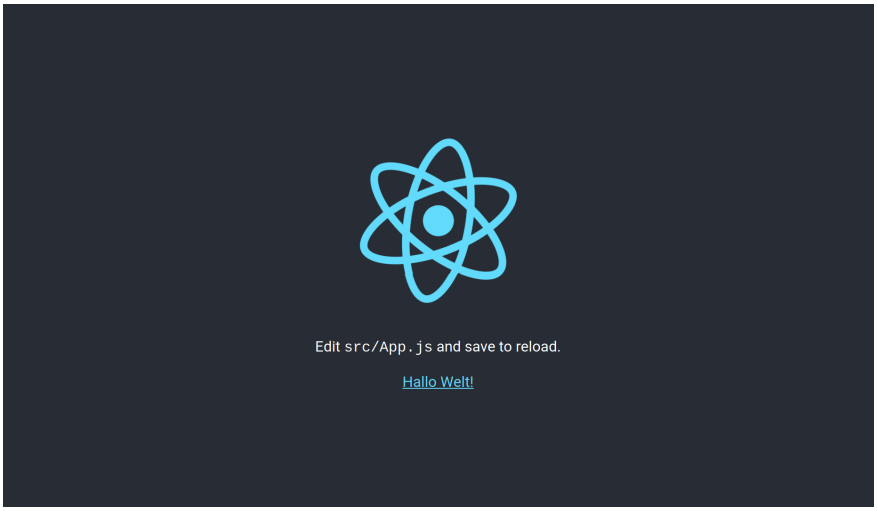
Um die Anwendung in Ihrem Browser anzuzeigen, öffnen Sie bitte http://localhost:3000/. Ihr Screen sollte nun wie folgt aus:

Im Folgenden Unterkapitel werden wir uns kurz den Entwicklungsserver anschauen.

Entwicklungsserver

`create-react-app` inkludiert einen webpack Entwicklungsserver. Dieser Server ermöglicht Ihnen das 'Hot Deployment' Ihres bearbeiteten Codes. Dies bedeutet, dass Ihre Code-Änderungen sofort im Browser sichtbar sind.

Aufgabe 1. Bearbeiten Sie `src/App.js` (beipielsweise wie im Code unten), um die Anwendung zu verändern.



```
import React from "react";
import logo from "./logo.svg";
import "./App.css";

function App() {
  return (
    <div className='App'>
      <header className='App-header'>
        <img src={logo} className='App-logo' alt='logo' />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className='App-link'
          href='https://reactjs.org'
          target='_blank'
          rel='noopener noreferrer'
        >
          Hallo Welt!
        </a>
      </header>
    </div>
  );
}

export default App;
```

Wie wir den GeoStyler und weitere notwendige Pakete für diese Anwendung installieren, werden wir uns im Folgenden genauer anschauen.

Dependencies hinzufügen

GeoStyler dependency hinzufügen

Um die `GeoStyler` dependency hinzuzufügen, navigieren Sie bitte (falls noch nicht geschehen) zum Ordner Ihres Projektes und führen Sie dort folgenden Befehl aus

```
npm install geostyler
```

> Für die Eingabe dieses Befehls müssen Sie ein neues Terminal öffnen und erneut zu Ihrem Ordner navigieren.

Dies fügt die neuste Version von `GeoStyler` zu Ihrer lokalen `package.json` Datei hinzu und lädt die Bibliothek in das Verzeichnis `node_modules`.

react-geo dependency hinzufügen

Analog gilt dies für die `react-geo` dependency:

```
npm i @terrestris/react-geo
```

Ant Design und OpenLayers dependencies hinzufügen

Sie haben vielleicht bemerkt, dass der Schritt von oben einige Warnungen hervorgerufen hat, die `GeoStyler` einschließen:

```
npm WARN geostyler@4.5.0 requires a peer of antd@3.x
npm WARN geostyler@4.5.0 requires a peer of ol@5.x bu
```

`npm` hat drei verschiedene Arten von dependencies:

`dependencies`

dependencies werden verwendet, um Pakete direkt zu spezifizieren, die zum *ausführen* des Codes Ihrer Anwendung benötigt werden (z.B. eine Frond-End Bibliothek wie Bootstrap).

`devDependecies`

devDependencies sind reserviert, um Pakete anzugeben, die zum *bauen* des Codes Ihrer Anwendung benötigt werden (z.B. test harnesses wie Jest oder Transpiler wie Babel).

`peerDependencies`

Unter bestimmten Bedingungen möchte man jedoch die *Kompatibilität* eines bestimmten Pakets mit dem Host-Paket audrücken. Npm bezeichnet diese dependency als peerDependencies. Normalerweise wird dies verwendet, um die Abhängigkeit eines Plugins innerhalb dieses oder eines ähnlichen Host-Pakets auszudrücken. In `react-geo` müssen wir `antd`, `ol` und `react` als *peer dependencies* aufgrund von scope Problemen definieren, da sie alle normalerweise vom Host-Paket/er Anwendung selbst in einer bestimmten Version referenziert wurden.

Da `npm` Abhängigkeiten hierarchisch behandelt, würde die doppelte Einbeziehung dieser Pakete in `react-geo` zu zwei verschiedenen *dependencies* führen, die in Ihrer Anwendung zur Laufzeit verfügbar sind. Um die Abhängigkeiten zwischen Ihrer Host-Anwendung und `react-geo` zu teilen, empfehlen wir `react-geo`, die vom Host-Paket gegebenen *dependencies* zu verwenden.

Um diese Anforderungen zu erfüllen, müssen wir die gewünschten *peer dependencies* mit folgendem Befehl installieren:

```
npm i antd@3.x ol@6
```

Jetzt sind wir bereit, alle `react-geo` Komponenten innerhalb unserer *geostyler-app* Anwendung zu verwenden.

Neben der grundlegenden React Anwendung werden wir im Folgenden Unterkapitel weitere Dateien für die CoVid-19 Map erstellen.

Hilfsdateien einbinden

Bevor Sie im Folgenden Kapitel eine interaktive Kartenanwendung erstellen, müssen zunächst weitere Dateien im *geostyler-app* Ordner erstellt werden.

Aufgabe 1. Erstellen Sie bitte zunächst folgende Dateien im *geostyler-app* Ordner (achten Sie hierbei dringend auf die Endungen):

- Attributions.js
- helper.js
- viewportHelper.js
- Workshop.css

> Da dieser Workshop den GeoStyler thematisiert, ist es nicht zwingend relevant
> den Aufbau und Inhalt des Ordners und der sich dort vorhandenen Skripte zu kennen. Hauptsächlich ist die `App.js` für diesen Workshop relevant.

Aufgabe 2. Kopieren Sie nun die jeweiligen Code-Blöcke (siehe unten) der entsprechenden Dateien und fügen Sie diese in die von Ihnen erstellte dazugehörige Datei ein.

Attributions.js

```
import React from 'react';

function Attributions() {
    return (
        <div className="attributions">
            <h1>Attributions</h1>
            <ul>
                <li><b>Countries:</b><a href="https://ahoceva
                <li><b>CoVid-19:</b><a href="https://github.
            </ul>
        </div>
    );
}

export default Attributions;
```

helper.js

```
import { Stroke, Fill, Style, Circle } from 'ol/style';
import VectorSource from 'ol/source/Vector';
import { Vector as VectorLayer } from 'ol/layer';
import { bbox as bboxStrategy } from 'ol/loadingstrategy';
import GeoJSON from 'ol/format/GeoJSON';

function getBaseLayer() {

    var baseSource = new VectorSource({
        format: new GeoJSON(),
        url: function (extent) {
            return 'https://ahocevar.com/geoserver/wfs?servi
                'version=1.1.0&request=GetFeature&typename=o
                'outputFormat=application/json&srsname=EPSG:
                'bbox=' + extent.join(',') + ',EPSG:3857';
        },
        strategy: bboxStrategy
    });

    var base = new VectorLayer({
        source: baseSource,
        style: getDefaultStyle(),
        projection: 'EPSG:3857'
    });

    return base;
}

function getCovidLayer(covidDeath) {
    var deathSource = new VectorSource({
        features: (new GeoJSON()).readFeatures(covidDeath, {
            featureProjection: 'EPSG:3857'
        })
    });

    var vector = new VectorLayer({
        source: deathSource,
        style: getDefaultStyle(),
        projection: 'EPSG:4326'
    });

    return vector;
}

function getDefaultStyle() {
    const defaultOlStyle = new Style({
        stroke: new Stroke({
```

```
            color: 'rgba(255, 255, 255, 1.0)',
            width: 1
        }),
        fill: new Fill({
            color: 'rgba(0, 0, 0, 1)'
        }),
        image: new Circle({
            fill: new Fill({
                color: 'rgba(255, 0, 0, 1.0)'
            }),
            radius: 5
        })
    });
    return defaultOlStyle;
}

export {
    getBaseLayer,
    getCovidLayer,
    getDefaultStyle
};
```

viewportHelper.js

```
function isElementInViewport (el) {

    var rect = el.getBoundingClientRect();

    return (
        (rect.top >= 0 && rect.top <= (window.innerHeight ||
        (rect.bottom >= 0 && rect.bottom <= (window.innerHei
    );
}

export default isElementInViewport;
```

workshop.css

```
html, body, #root, .App, #map {
    margin: 0;
    padding: 0;
    height: calc(100vh + (3 * 30vh) + (2 * 130vh) - 30vh);
}

.App {
    background-color: black;
}

#map {
    position: fixed;
    height: 100vh;
    width: 100vw;
}

.ws-toggle-editor-btn {
    position: fixed;
    z-index: 9999;
}

.ws-overlay {
    position: relative;
    border: 2px solid #f2f51e;
    border-radius: 5px;
    width: 30vw;
    height: 30vh;
    z-index: 1000;
    background-color: rgba(255, 255, 255, 0.25);
}

.ws-overlay, .ws-overlay > h1 {
    color: white;
}

#ws-overlay-1 {
    display: inline-block;
    margin-left: 5vw;
    margin-bottom: calc(130vh);
    border-color: #f2f51e;
}

#ws-overlay-2 {
    margin-left: calc(100vw - 30vw - 5vw);
    margin-bottom: calc(130vh);
    border-color: #52c41a;
}
```

```
#ws-overlay-3 {
    margin-left: 5vw;
    margin-bottom: calc(100vh - 30vh);
    border-color: #4464ff;
}

.filter-editor-window, .rule-generator-window, .symbolizer-e
    z-index: 1000 !important;
    position: fixed !important;
    left: 60vw !important;
    top: 20vh !important;
    transform: none !important;
}

.attributions {
    position: relative;
}

.ant-drawer-content-wrapper {
    height: 400px !important;
}
```

Wenn alle Dateien erstellt wurden, dann können wir endlich mit der Erstellung der Webmap beginnen. Dies schauen wir uns im nächsten Unterkapitel an.

Karte hinzufügen

Nachdem wir die für diesen Workshop notwendigen Dependencies installiert haben, können wir deren Komponenten in der *geostyler-app* Anwendung nutzen. Als nächstes werden wir react-geos Map Komponente mit der Hintergrundkarte der Anwendung hinzufügen.

> Falls Sie mehr über react-geo herausfinden möchten, dann können Sie hier an dem react-geo-Workshop teilnehmen.

Bevor React, sowie OpenLayers und react-geo verwendet werden können, müssen diese zu Beginn der Anwendung aus der jeweiligen Bibliothek importiert werden. Dies geschieht beispielsweise durch:

```
import OlMap from "ol/Map";
import { MapComponent } from "@terrestris/react-geo";

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base],
  interactions: [new DragPan()]
});
```

Darüber hinaus muss hier eine `map` Variable erstellt werden, welche grundlegende Einstellungen der Karteninstanz beinhaltet, wie z.B. das Zoomlevel, die Projektion oder auch die Layer.

Die react-geo Komponente (`MapComponent`) dient als Wrapper für eine OpenLayers map. Die `OlMap` Variable `map` wird der `MapComponent` als eine Property (prop) hinzugefügt.

```
<MapComponent map={map} />
```

Die Hintergrundkarte (im Code mit der Variable `base` genannt) ist ein OpenLayers Layer und wird durch den Befehl `import { getBaseLayer} from './helper';` der App.js Datei zugänglich gemacht.

> Der Übersicht halber wurde diese Datei von `App.js` nach `helper.js` ausgelagert.

Aufgabe 1. Öffnen Sie einen Texteditor Ihrer Wahl (falls noch nicht geschehen) und öffnen Sie die Datei `App.js` von dem `src` Ordner Ihrer *geostyler-app* Anwendung. Erstellen Sie nun eine `map` - Variable und fügen Sie diese mit dem `base` - Layer der Applikation hinzu.

Folgendermaßen wird die Karte bei erfolgreicher Bearbeitung der Aufgabe (und Speichern der Änderungen) im Browser angezeigt:

Der Code Ihrer Lösung könnte wie folgt aussehen:

```
import React from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import { getBaseLayer } from "./helper";

import { MapComponent } from "@terrestris/react-geo";

var base = getBaseLayer();

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base],
  interactions: [new DragPan()]
});

function App() {
  return (
    <div className='App'>
      <MapComponent map={map} />
    </div>
  );
}

export default App;
```

Im nächsten Unterkapitel werden wir der Anwendung drei Boxen hinzufügen, welche den Style des Layers (s. Kapitel 2.7), abhängig von der aktuell im Viewport zu sehenden Box verändert.

Infoboxen hinzufügen

In diesem Abschnitt werden wir der Anwendung drei Boxen (div-Elemente, die durch Angabe von width & height wie eine Box aussehen) hinzufügen, welche mit einem *scroll eventlistener* versehen sind. Ziel ist es, den Style des Layers (welcher im folgenden Kapitel hinzugefügt wird) abhängig von der aktuell im Viewport zu sehenden Box zu verändern.

Die jeweiligen Boxen wurden wie folgt erstellt (hier beispielhaft die zweite Box):

```
<div id='ws-overlay-2' className='ws-overlay'>
  <h1>Overlay {visibleBox + 1}</h1>
  <p>Put your info text here</p>
</div>
```

Der `App.js` Datei wird durch den Befehl `import isElementInViewport from "./viewportHelper";` eine Funktion zugänglich gemacht, welche in der `viewportHelper.js` Datei definiert ist.

Diese Funktion erkennt die sich aktuell im Viewport befindende Box.

> Für diesen Workshop reicht es zu wissen, dass es diese Funktion gibt. Eine genaue Erläuterung des sich in der `viewportHelper.js` Datei befindenden Codes ist für diesen Workshop somit nicht relevant.

Darüber hinaus werden innerhalb der `App` - Funktion zwei State Variablen hinzugefügt:

```
let [visibleBox, setVisibleBox] = useState(0);
```

Diese werden im Folgenden benötigt, um den jeweiligen aktuellen State, ausgehend von der sich aktuell im Viewport befindenden Box, zu verändern.

Aufgabe 1. Erstellen Sie drei Infoboxen und fügen Sie diese in die `return` - Funktion der `App` - Funktion ein. Diese sollten jeweils die id `'ws-overlay-1'` (Zahl abhängig von der Box) und die Klasse (`className`) `ws-overlay` beinhalten. Des Weiteren soll sich in der `h1` folgender Code befinden: `Overlay {visibleBox + 1}`

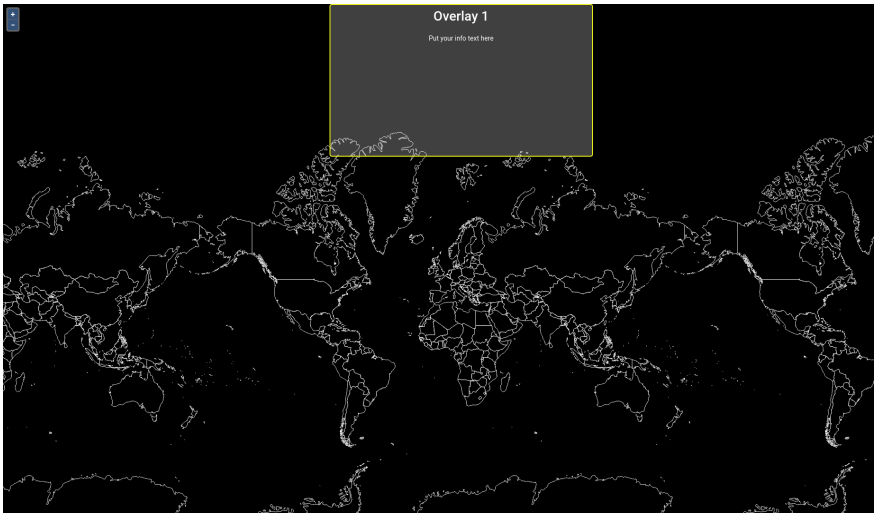
Aufgabe 2. Fügen Sie zwei State Variablen mit den Namen `visibleBox` und `setVisibleBox` der *function Component* hinzu und weisen Sie diesen den Wert `useState(0)` zu.

Aufgabe 3. Fügen Sie folgenden Code Block unterhalb der in Aufgabe 2 erstellten Variablen ein.

```
useEffect(() => {
  // add scroll eventlistener
  // unfortunately, this will be re-run as soon as visible
  // box changes. Otherwise we don't have visible box in ou
  const getVisibleBox = () => {
    const boxes = [
      document.getElementById("ws-overlay-1"),
      document.getElementById("ws-overlay-2"),
      document.getElementById("ws-overlay-3")
    ];
    const boxIdx = boxes.findIndex(box => isElementInViewpo
    return boxIdx >= 0 ? boxIdx : visibleBox;
  };

  const handleScroll = () => {
    const newVisibleBox = getVisibleBox();
    if (newVisibleBox !== visibleBox) {
      setVisibleBox(newVisibleBox);
    }
  };

  document.addEventListener("scroll", handleScroll);

  handleScroll();

  return () => {
    document.removeEventListener("scroll", handleScroll);
  };
}, [visibleBox]);
```

> Für diesen Workshop reicht es zu wissen, dass es die `useEffect()` - Funktion, sowie den `addEventListener` gibt. Eine genaue Erläuterung ist für diesen Workshop somit nicht relevant.

Ihre Anwendung sollte, insofern Sie die Datein gespeichert haben, nun wie folgt aussehen:


Overlay 1
Put your info text here

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getBaseLayer } from "./helper";

import { MapComponent } from "@terrestris/react-geo";

var base = getBaseLayer();

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base],
  interactions: [new DragPan()]
});

function App() {
  let [visibleBox, setVisibleBox] = useState(0);

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in ou
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewpo
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };
```

```
    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <MapComponent map={map} />
      <span id='ws-overlay-1' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </span>
      <div id='ws-overlay-2' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </div>
      <div id='ws-overlay-3' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </div>
      <Attributions />
    </div>
  );
}

export default App;
```

Im nächsten Unterkapitel werden wir der Karte einen Layer hinzufügen.

CoVid-Layer hinzufügen

In diesem Unterkapitel werden wir der `map` Variable einen weiteren Layer zuweisen (neben dem bereits im Kapitel 2.5 hinzugefügten base-Layer).

Dafür wird der covidDeath Datensatz zunächst importiert (1), der `vector` Variable zugewiesen und in der `map` Variable referenziert (2).

(1)

```
import covidDeath from "./data/covid-death.json";
var vector = getCovidLayer(covidDeath);
```

> Die Funktion `getCovidLayer()` stammt ebenfalls aus der `helper.js` Datei. Auch hier ist dieser Schritt für das Verständnis des GeoStylers nicht notwenig. Auf eine genauere Erklärung wird hier somit verzichtet.

(2)

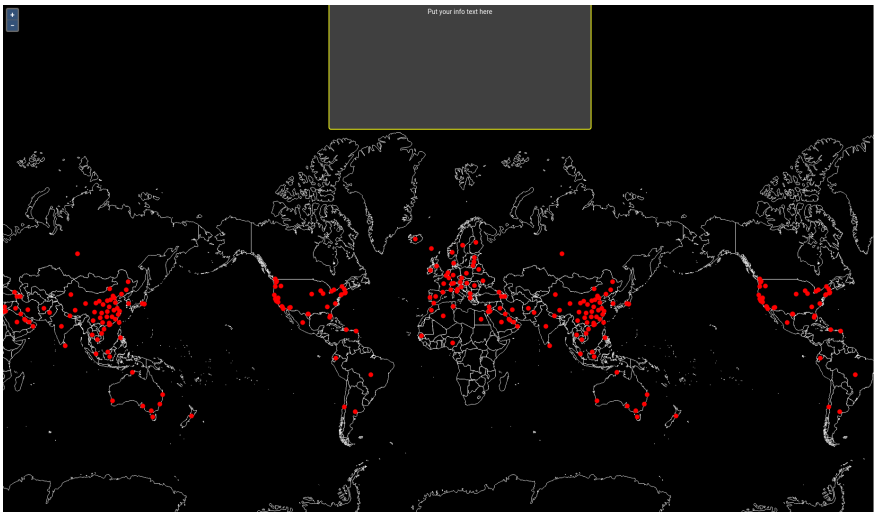
```
const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});
```

Aufgabe 1. Erstellen Sie nun einen neuen Ordner mit dem Namen `data` innerhalb des `src` -Ordners.

Aufgabe 2. Speichern Sie den Inhalt dieser JSON-Datei (hier) und fügen Sie diesen anschließend in eine Datei mit dem Namen `covid-death.json` in den `data` Ordner ein.

Aufgabe 3. Importieren Sie nun das GeoJSON in die `App.js` Datei und erstellen Sie daraus einen OL-Layer mit dem Namen `vector` , mit Hilfe der Funktion `getCovidLayer` . Fügen Sie die `vector` - Variable nun dem `map` - Objekt hinzu.

Die Anwendung sollte anschließend wie folgt aussehen:



Der Code Ihrer Lösung könnte wie folgt aussehen:

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getBaseLayer, getCovidLayer } from "./helper";

import { MapComponent } from "@terrestris/react-geo";

import covidDeath from "./data/covid-death.json";

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

function App() {
  let [visibleBox, setVisibleBox] = useState(0);

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in ou
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewpo
```

```
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <MapComponent map={map} />
      <span id='ws-overlay-1' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </span>
      <div id='ws-overlay-2' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </div>
      <div id='ws-overlay-3' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </div>
      <Attributions />
    </div>
  );
}

export default App;
```

Im folgenden Unterkapitel werden wir der Anwendung einen Drawer und einen Button hinzufügen. Innerhalb dieses Drawers werden wir dann folglich den `GeoStyler` einbinden.

Drawer und Toggle hinzufügen

In diesem Unterkapitel wird ein Button und ein Drawer von antd importiert. Der Button sorgt dafür, dass sich der Drawer per toggle ein- und ausblenden lässt.

```
<Drawer
  title='GeoStyler Editor'
  placement='top'
  closable={true}
  onClose={() => {
    setDrawerVisible(false);
  }}
  visible={drawerVisible}
  mask={false}
></Drawer>
```

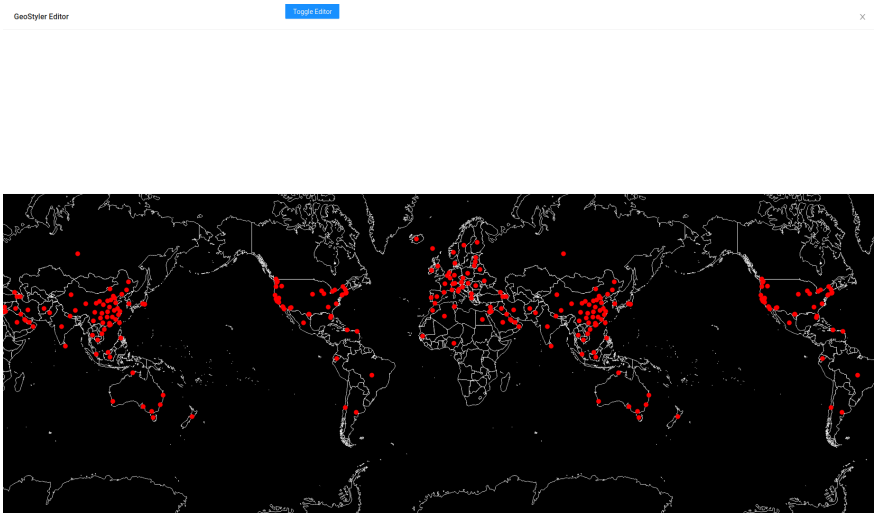
Wie anhand des folgenden Code-Auszuges zu erkennen ist, geschieht der toggle-Effekt durch die Veränderung des aktuellen States (`currentState => !currentState`). Der Button und Drawer sind hierbei durch die `setDrawerVisible()` - Funktion vernüpft.

```
  <Button
    className='ws-toggle-editor-btn'
    type='primary'
    onClick={() => {
      setDrawerVisible(currentState => !currentState);
     }}
  >
    Toggle Editor
  </Button>
```

> Hier finden Sie weitere Informationen bezüglich State und setState().

Aufgabe 1. Fügen Sie zwei State Variablen mit den Namen `drawerVisible` und `setDrawerVisible` der function Component hinzu und weisen Sie diesen den Wert `useState(false)` zu.

Aufgabe 2. Erstellen Sie jeweils einen `Drawer` und einen `Button` mit den oben dargestellten Eigenschaften und fügen Sie diese der `return` - Funktion der `App` - Funktion hinzu.



Per Klick auf den Button öffnet sich folglich der Drawer (in der oberen Abbildung bereits geöffnet).

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";
import { Drawer, Button } from "antd";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getBaseLayer, getCovidLayer } from "./helper";

import { MapComponent } from "@terrestris/react-geo";

import covidDeath from "./data/covid-death.json";

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

function App() {
  let [drawerVisible, setDrawerVisible] = useState(false);
  let [visibleBox, setVisibleBox] = useState(0);

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in ou
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
```

```
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewp
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <Button
        className='ws-toggle-editor-btn'
        type='primary'
        onClick={() => {
          setDrawerVisible(currentState => !currentState);
        }}
      >
        Toggle Editor
      </Button>
      <MapComponent map={map} />
      <Drawer
        title='GeoStyler Editor'
        placement='top'
        closable={true}
        onClose={() => {
          setDrawerVisible(false);
        }}
        visible={drawerVisible}
        mask={false}
      ></Drawer>
      <span id='ws-overlay-1' className='ws-overlay'>
        <h1>Overlay {visibleBox + 1}</h1>
        <p>Put your info text here</p>
      </span>
      <div id='ws-overlay-2' className='ws-overlay'>
```

```
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <div id='ws-overlay-3' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <Attributions />
      </div>
    );
}

export default App;
```

Das nächste Überkapitel beschäftigt sich mit dem GeoStyler und geht dabei auf die grundlegende Verwendung der UI, des Daten Parsers und des Style Parsers ein.

GeoStyler einbinden

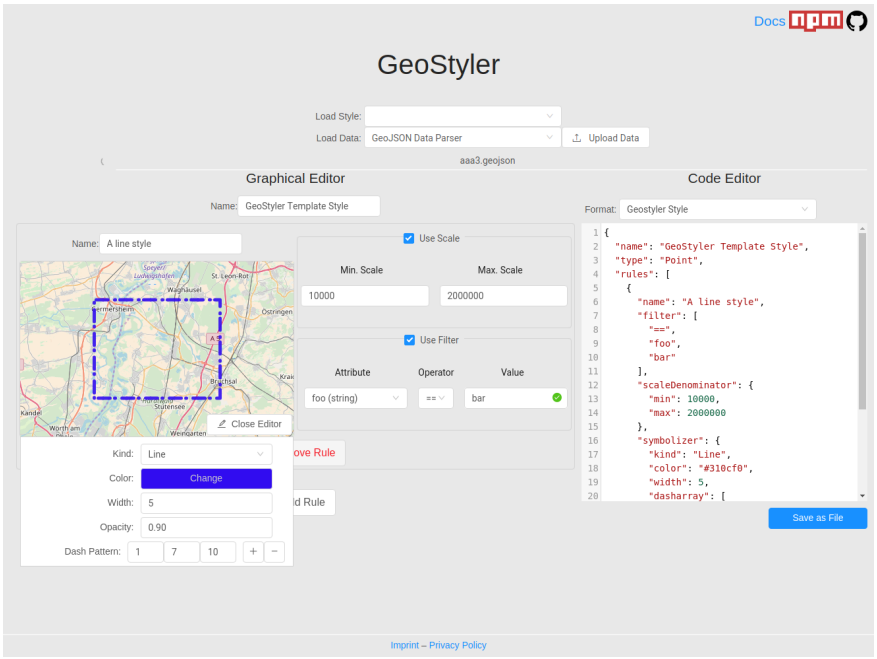
Im folgenden Überkapitel werden Sie endlich GeoStyler der Anwendung hinzufügen.

Zunächst wird jedoch kurz der Aufbau und die Funktionsweise des GeoStylers erläutert.

Das kartographische Stylen von Geodaten im Web ist seit Jahren ein wiederkehrendes Thema in der Geoinformatik-Welt. Es existieren verschiedenste Standards – Offizielle Standards, z.B. OGC Styled Layer Descriptor mit OGC Filter Encoding sowie Industriestandards, beispielsweise Mapbox Styles und projektbezogene Styling-Vorschriften, z.B. in QGIS oder OpenLayers.

Es fehlt jedoch eine interaktive webbasierte Software, um Anwender in die Lage zu versetzen die kartographische Ausgestaltung ihrer Geodaten auf einfache Weise zu erledigen. Es gibt zwar vereinzelte Lösungen für einzelne der oben genannten Standards, eine gesamtheitliche Web-Oberfläche, um unter anderem auch Styling-Vorschriften in diverse Formate zu überführen, fehlte bislang.

Unter dem Projektnamen „GeoStyler“ entsteht aktuell ein webbasiertes Werkzeug zur interaktiven Erstellung von kartographischen Style-Vorschriften für Geodaten.



Grundsätzlich besteht der GeoStyler aus fünf Hauptkomponenten:

1. GeoStyler: React basierte Komponenten Bibliothek, die es ermöglicht eine individuell angepasste GUI zum Editieren und Parsen kartographischer Styles zu entwickeln.
2. GeoStyler-Style: Definition des in GeoStyler intern genutzten Styles. Die Nutzung eines zentralen Styles ermöglicht es von einem beliebigen externen Style zu jedem anderen externen Style zu parsen. Vorraussetzung dafür ist lediglich die Implementierung eines entsprechenden Style Parsers (siehe Punkt 4).
3. Geostyler-Data: Definition der in GeoStyler intern genutzten Datenstruktur.
4. Style Parser: Implementierungen zum Parsen existierender Style-Vorschriften von/zu Geostyler-Style. Beispiele: geostyler-sld-parser und geostyler-openlayers-parser
5. Data Parser: Implementierungen zum Parsen zu unterstützender Geodaten-Formate.

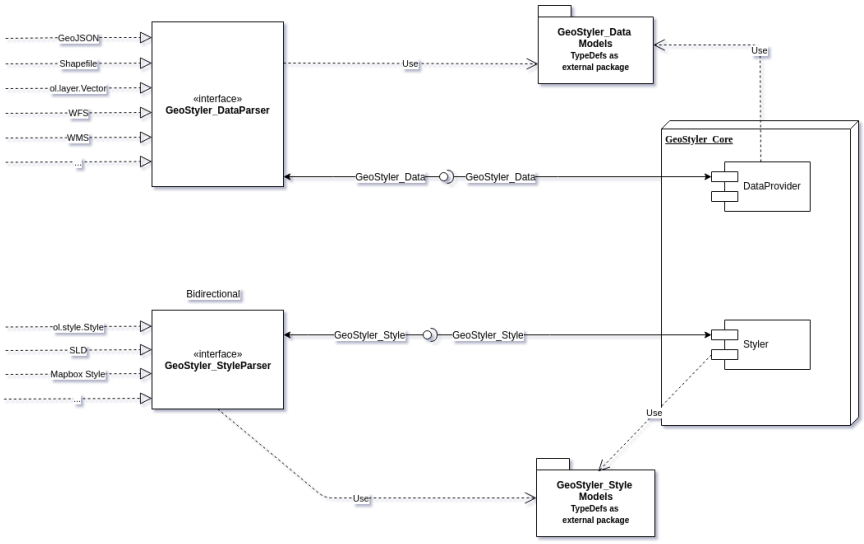
Aktuell können folgende Formate genutzt werden: Style-Vorschriften

- OGC SLD
- OpenLayers Styles

Geodaten-Formate

- GeoJSON
- OGC WFS

GeoStyler wird als Open Source Projekt realisiert und die offene Architektur ermöglicht es sehr einfach weitere Formate (sowohl für Style-Vorschriften als auch für Geodaten) durch Implementierung entsprechender Parser in GeoStyler zu integrieren.



Für weitere Informationen bezüglich des GeoStylers besuchen Sie gerne die offzielle GeoStyler-Github Seite.

UI Komponente hinzufügen

In diesem Unterkapitel wird der GeoSstyler der Anwendung zugänglich gemacht.

Die anschließende Einbindung in die Anwendung erfolgt durch den Import:

```
import { Style as GsStyle } from "geostyler";
```

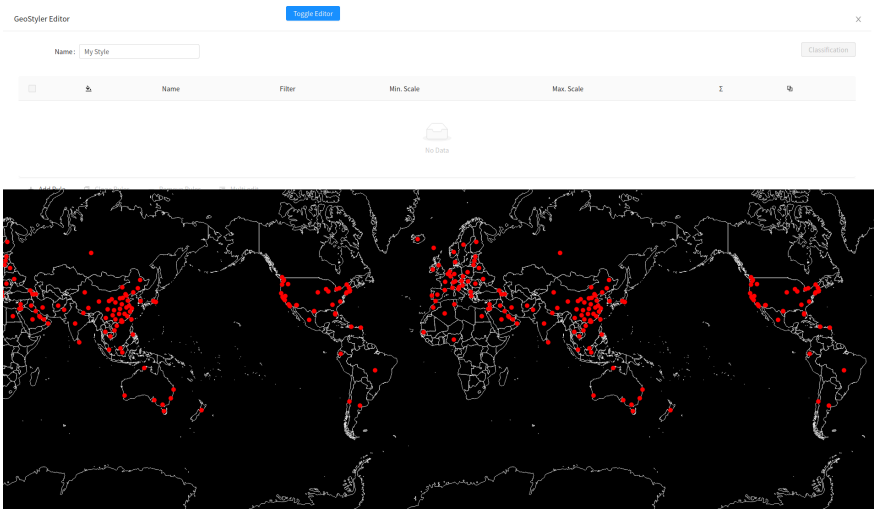
Innerhalb des Drawers, welcher in Kapitel 2.8 hinzugefügt wurde, befindet sich die oben importierte `<GsStyle />` Komponente (s. folgendender Code-Block). Durch einen Klick auf den Button wird somit der Drawer und der GeoStyler sichtbar.

```
<Drawer
  title='GeoStyler Editor'
  placement='top'
  closable={true}
  onClose={() => {
    setDrawerVisible(false);
  }}
  visible={drawerVisible}
  mask={false}
>
  <GsStyle compact={true} />
</Drawer>
```

Durch die Einbindung des GeoStylers ist im Grund alles für die Inbetriebnahme gegeben, jedoch fehlt hierbei noch die Verknüpfung mit der Applikation und den Daten. Der Anwendung bedarf es nun den notwendigen Parsern, um SLD bspw. in OpenLayers Styles zu überführen. Darauf wird in den folgenden Kapiteln eingegangen.

Aufgabe 1. Importieren Sie den `GeoStyler` und fügen diese in den Drawer ein.

Die Applikation sollte nun wie folgt aussehen:



Der Code Ihrer Lösung könnte wie folgt aussehen:

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";
import { Drawer, Button } from "antd";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getBaseLayer, getCovidLayer } from "./helper";

import { MapComponent } from "@terrestris/react-geo";

import { Style as GsStyle } from "geostyler";

import covidDeath from "./data/covid-death.json";

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

function App() {
  let [drawerVisible, setDrawerVisible] = useState(false);
  let [visibleBox, setVisibleBox] = useState(0);

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in ou
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
```

```
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewp
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <Button
        className='ws-toggle-editor-btn'
        type='primary'
        onClick={() => {
          setDrawerVisible(currentState => !currentState);
        }}
      >
        Toggle Editor
      </Button>
      <MapComponent map={map} />
      <Drawer
        title='GeoStyler Editor'
        placement='top'
        closable={true}
        onClose={() => {
          setDrawerVisible(false);
        }}
        visible={drawerVisible}
        mask={false}
      >
        <GsStyle compact={true} />
      </Drawer>
      <span id='ws-overlay-1' className='ws-overlay'>
```

```
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </span>
        <div id='ws-overlay-2' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <div id='ws-overlay-3' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <Attributions />
      </div>
    );
}

export default App;
```

Im nächsten Unterkapitel werden wir einen `default Style` einbinden und einen Parser aus der `GeoStyler` Bibliothek importieren.

Style Parser verwenden

In diesem Unterkapitel kommt erstmals ein Parser zum Einsatz. Dieser wird direkt von der GeoStyler Bibliothek importiert und in der Variable `olParser` referenziert (s. folgender Code-Block). Der Parser wird verwendet, um von bereits existierenden Style-Vorschriften zu GeoStyler-Style zu transformieren.

```
import OpenLayersParser from "geostyler-openlayers-parser";
const olParser = new OpenLayersParser();

OlParser
  .readStyle(defaultOlStyle)
  .then(gsStyle => console.log(gsStyle))
  .catch(error => console.log(error));
```

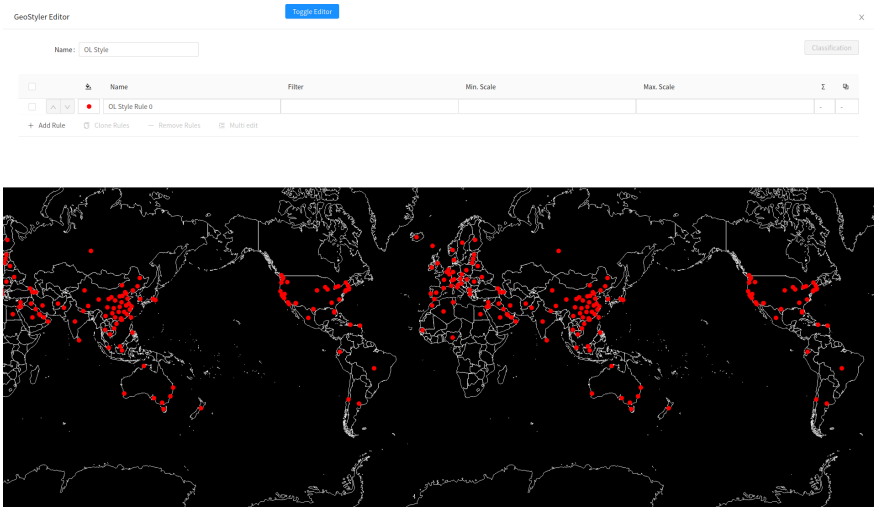
Desweiteren wird ein *defaultOlStyle* definiert (ebenfalls aus der `helper.js` Datei importiert).

```
import { getDefaultStyle } from "./helper";
const defaultOlStyle = getDefaultStyle();
```

Der `OpenLayersParser` ermöglicht hierbei, dass die Style-Vorschriften in den GeoStyler-Style überführt werden. Das `defaultOlStyle` kann somit geparst werden.

Der `GeoStyler` nimmt nun einen Stil an, jedoch verändert sich dieser beim Scrollen noch nicht. Diese Verknüpfung mit den Daten erfolgt erst im kommenden Kapitel.

Die Applikation sollte nun wie folgt aussehen:



Der Code Ihrer Lösung könnte wie folgt aussehen:

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";
import { Drawer, Button } from "antd";

import OpenLayersParser from "geostyler-openlayers-parser";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getDefaultStyle, getBaseLayer, getCovidLayer } from

import { MapComponent } from "@terrestris/react-geo";

import { Style as GsStyle } from "geostyler";

import covidDeath from "./data/covid-death.json";

const defaultOlStyle = getDefaultStyle();

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

const olParser = new OpenLayersParser();

function App() {
  let [styles, setStyles] = useState([]);
  let [drawerVisible, setDrawerVisible] = useState(false);
  let [visibleBox, setVisibleBox] = useState(0);
```

```
  useEffect(() => {
    // on page init parse default style once
    // and setup the styles array
    olParser
      .readStyle(defaultOlStyle)
      .then(gsStyle => {
        const newStyles = [];
        for (var i = 0; i < 3; i++) {
          newStyles.push(JSON.parse(JSON.stringify(gsStyle))
        }
        setStyles(newStyles);
      })
      .catch(error => console.log(error));
  }, []);

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in ou
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewpo
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <Button
```

```
          className='ws-toggle-editor-btn'
          type='primary'
          onClick={() => {
            setDrawerVisible(currentState => !currentState);
          }}
        >
          Toggle Editor
        </Button>
        <MapComponent map={map} />
        <Drawer
          title='GeoStyler Editor'
          placement='top'
          closable={true}
          onClose={() => {
            setDrawerVisible(false);
          }}
          visible={drawerVisible}
          mask={false}
        >
          <GsStyle
            style={styles[visibleBox]}
            compact={true}
            onStyleChange={newStyle => {
              setStyles(oldStyles => {
                const newStyles = JSON.parse(JSON.stringify(old
                newStyles[visibleBox] = newStyle;
                return newStyles;
              });
            }}
          />
        </Drawer>
        <span id='ws-overlay-1' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </span>
        <div id='ws-overlay-2' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <div id='ws-overlay-3' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <Attributions />
      </div>
    );
}
```

```
export default App;
```

Wie sich der Layer abhängig von der aktuell im Viewport zu sehenden Box verändert, werden wir im nächsten Kapitel sehen.

Layer - Geostyler Verknüpfung

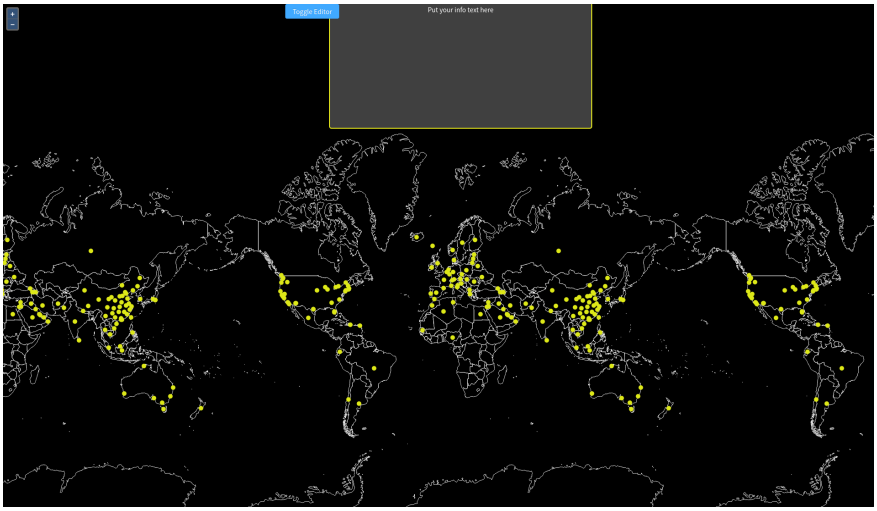
In diesem Unterkapitel wird der Geostyler mit dem Layer verküpft.

Darüber hinaus wird durch folgende Funktion die Karte jedes mal geupdated, wenn sich entweder eine andere Box im Viewport befindet, oder der Style geändert wird.

```
useEffect(() => {
  // update the map layer when either visibleBox or styles
  var newStyle = styles[visibleBox];
  if (newStyle) {
    olParser
      .writeStyle(newStyle)
      .then(olStyle => {
        vector.setStyle(olStyle);
      })
      .catch(error => console.log(error));
  }
});
```

Aufgabe 1. Fügen sie die oben dargestellte Funktion der Anwendung hinzu.

Aufgabe 2. Ändern Sie nun den Stil und Scrollen Sie hoch und runter (soweit, bis eine andere Box im Viewport sichtbar wird). Welche Veränderung wird sichtbar?



Wie Sie anhand der Abbildung erkennen können, wurde die Farbe des Layers verändert. Wenn Sie hoch und runter scrollen ändert sich der Style abhängig davon, welche Box aktuell im Viewport sichtbar ist.

Der Code Ihrer Lösung könnte wie folgt aussehen:

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";
import { Drawer, Button } from "antd";

import OpenLayersParser from "geostyler-openlayers-parser";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getDefaultStyle, getBaseLayer, getCovidLayer } from

import { MapComponent } from "@terrestris/react-geo";

import { Style as GsStyle } from "geostyler";

import covidDeath from "./data/covid-death.json";

const defaultOlStyle = getDefaultStyle();

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

const olParser = new OpenLayersParser();

function App() {
  let [styles, setStyles] = useState([]);
  let [drawerVisible, setDrawerVisible] = useState(false);
  let [visibleBox, setVisibleBox] = useState(0);
```

```
  useEffect(() => {
    // on page init parse default style once
    // and setup the styles array
    olParser
      .readStyle(defaultOlStyle)
      .then(gsStyle => {
        const newStyles = [];
        for (var i = 0; i < 3; i++) {
          newStyles.push(JSON.parse(JSON.stringify(gsStyle))
        }
        setStyles(newStyles);
      })
      .catch(error => console.log(error));
  }, []);

  useEffect(() => {
    // update the map layer when either visibleBox or styles
    var newStyle = styles[visibleBox];
    if (newStyle) {
      olParser
        .writeStyle(newStyle)
        .then(olStyle => {
          vector.setStyle(olStyle);
        })
        .catch(error => console.log(error));
    }
  });

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in ou
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewpo
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };
```

```
    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <Button
        className='ws-toggle-editor-btn'
        type='primary'
        onClick={() => {
          setDrawerVisible(currentState => !currentState);
        }}
      >
        Toggle Editor
      </Button>
      <MapComponent map={map} />
      <Drawer
        title='GeoStyler Editor'
        placement='top'
        closable={true}
        onClose={() => {
          setDrawerVisible(false);
        }}
        visible={drawerVisible}
        mask={false}
      >
        <GsStyle
          style={styles[visibleBox]}
          compact={true}
          onStyleChange={newStyle => {
            olParser
              .writeStyle(newStyle)
              .then(olStyle => {
                vector.setStyle(olStyle);
              })
              .catch(error => console.log(error));
            setStyles(oldStyles => {
              const newStyles = JSON.parse(JSON.stringify(ol
              newStyles[visibleBox] = newStyle;
              return newStyles;
            });
          }}
```

```
          />
        </Drawer>
        <span id='ws-overlay-1' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </span>
        <div id='ws-overlay-2' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <div id='ws-overlay-3' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <Attributions />
      </div>
    );
}

export default App;
```

Im letzten Unterkapitel werden wir einen weiteren Parser (Daten Parser) hinzufügen, welcher es erlaubt, die UI mit Informationen der importierten Daten zu füllen.

Daten Parser verwenden

In diesem Unterkapitel wird der `GeoJSONParser`, welchen wir mit dem Befehl

```
import GeoJSONParser from "geostyler-geojson-parser";
```

der Anwendung zugänglich machen, eingebunden. Dieser Datenparser wird verwendet, um die UI mit Informationen aus den importierten Daten zu füllen. Der Parser ermöglicht, dass sich Änderungen im Layer auf den Stil und Änderungen im Stil auf den Layer auswirken.

Aufgabe 1. Importieren Sie den GeoJSONParser (analog zu dem bereits importierten OpenLayersParser) und erstellen Sie eine neue GeoJSONParser Instanz mit dem Namen `geojsonParser`.

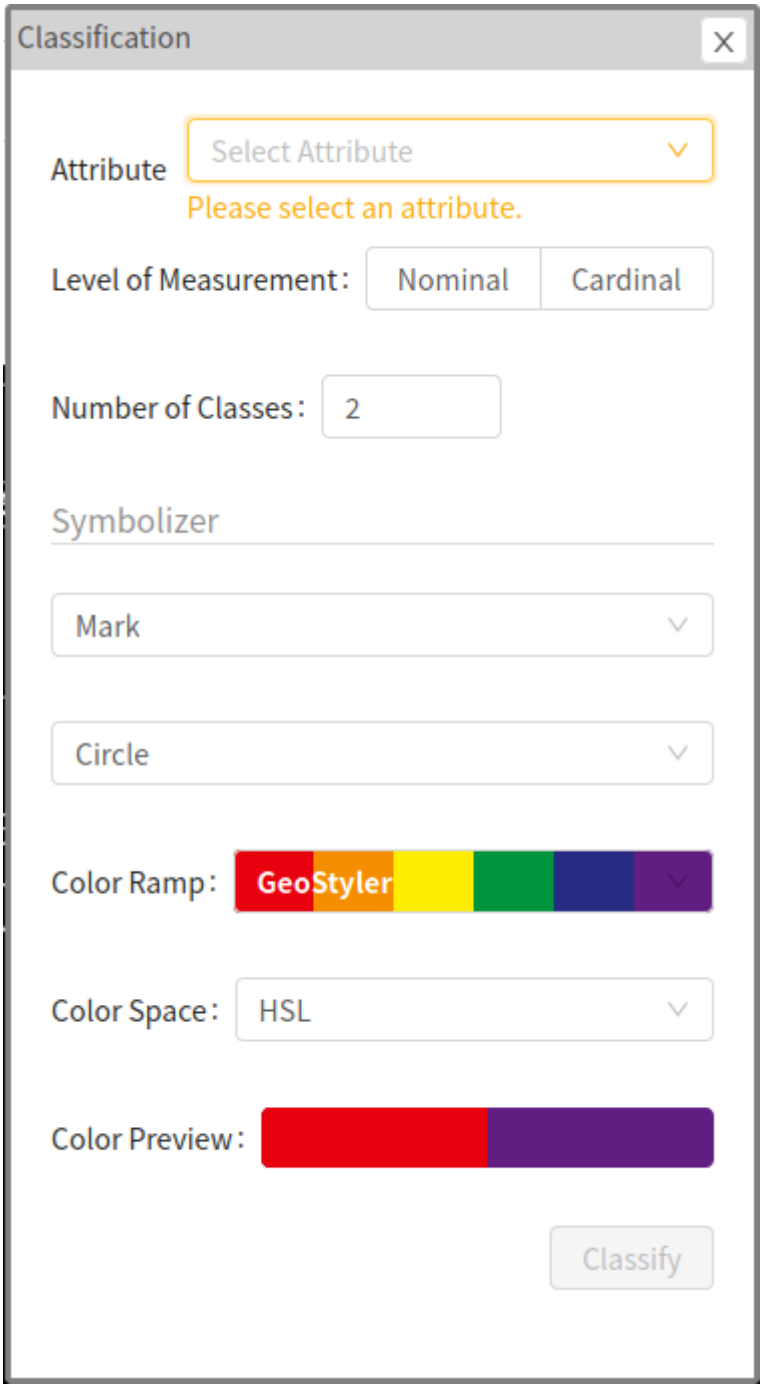
Aufgabe 2. Erstellen Sie eine weitere `useEffect()` - Funktion mit dem `geojsonParser` und lassen Sie diesen den `covidDeath` - Layer lesen (`.readData()`).

Aufgabe 3. Fügen Sie anschließend folgenden Code Block unterhalb der `.readData()` - Methode ein:

```
      .then(gsData => {
        setData(gsData);
      })
      .catch(error => console.log(error));
  }, [data]);
```

Dieser Code Block sorgt dafür, dass die Daten des Layers, durch den Daten Parser in der UI angezeigt werden.

Aufgabe 4. Klicken Sie nun im `GeoStyler` auf *Classification* und folglich auf *Attribute*. Schauen Sie sich analog dazu die `covid-death.json` Datei an. Was fällt Ihnen auf?



Der Code Ihrer Lösung könnte wie folgt aussehen:

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";
import { Drawer, Button } from "antd";

import OpenLayersParser from "geostyler-openlayers-parser";
import GeoJSONParser from "geostyler-geojson-parser";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getDefaultStyle, getBaseLayer, getCovidLayer } from

import { MapComponent } from "@terrestris/react-geo";

import { Style as GsStyle } from "geostyler";

import covidDeath from "./data/covid-death.json";

const defaultOlStyle = getDefaultStyle();

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

const olParser = new OpenLayersParser();
const geojsonParser = new GeoJSONParser();

function App() {
  let [styles, setStyles] = useState([]);
  let [drawerVisible, setDrawerVisible] = useState(false);
```

```
  let [visibleBox, setVisibleBox] = useState(0);
  let [data, setData] = useState();

  useEffect(() => {
    // on page init parse default style once
    // and setup the styles array
    olParser
      .readStyle(defaultOlStyle)
      .then(gsStyle => {
        const newStyles = [];
        for (var i = 0; i < 3; i++) {
          newStyles.push(JSON.parse(JSON.stringify(gsStyle))
        }
        setStyles(newStyles);
      })
      .catch(error => console.log(error));
  }, []);

  useEffect(() => {
    // parse data as soon as it changes
    geojsonParser
      .readData(covidDeath)
      .then(gsData => {
        setData(gsData);
      })
      .catch(error => console.log(error));
  }, [data]);

  useEffect(() => {
    // update the map layer when either visibleBox or styles
    var newStyle = styles[visibleBox];
    if (newStyle) {
      olParser
        .writeStyle(newStyle)
        .then(olStyle => {
          vector.setStyle(olStyle);
        })
        .catch(error => console.log(error));
    }
  });

  useEffect(() => {
    // add scroll eventlistener
    // unfortunately, this will be re-run as soon as visible
    // box changes. Otherwise we don't have visible box in o
    const getVisibleBox = () => {
      const boxes = [
        document.getElementById("ws-overlay-1"),
```

```
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewp
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <Button
        className='ws-toggle-editor-btn'
        type='primary'
        onClick={() => {
          setDrawerVisible(currentState => !currentState);
        }}
      >
        Toggle Editor
      </Button>
      <MapComponent map={map} />
      <Drawer
        title='GeoStyler Editor'
        placement='top'
        closable={true}
        onClose={() => {
          setDrawerVisible(false);
        }}
        visible={drawerVisible}
        mask={false}
      >
        <GsStyle
          style={styles[visibleBox]}
          compact={true}
```

```
            data={data}
            onStyleChange={newStyle => {
              olParser
                .writeStyle(newStyle)
                .then(olStyle => {
                  vector.setStyle(olStyle);
                })
                .catch(error => console.log(error));
              setStyles(oldStyles => {
                const newStyles = JSON.parse(JSON.stringify(ol
                newStyles[visibleBox] = newStyle;
                return newStyles;
              });
            }}
          />
        </Drawer>
        <span id='ws-overlay-1' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </span>
        <div id='ws-overlay-2' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <div id='ws-overlay-3' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <Attributions />
      </div>
    );
}

export default App;
```

Zusammenfassung

Glückwunsch! Sie haben den Geostyler Workshop beendet!

In diesem Workshop haben Sie gelernt wie Sie Ihre erste react-basierte Webanwendung mit Geostyler Funktionalitäten errichten. Durch das Hinzufügen der react-geo Komponente `MapComponent`, sowie der Einbindung von Geostyler inklusive deren Parser haben Sie (einige) Möglichkeiten des Geostylers kennengelernt, um damit Geodaten webbasiert nach Ihren Bedürfnissen graphisch ansprechend darzustellen.

Und hier ist der vollständige Source-Code.

```
import React, { useState, useEffect } from "react";

import OlMap from "ol/Map";
import OlView from "ol/View";
import DragPan from "ol/interaction/DragPan";
import { Drawer, Button } from "antd";

import OpenLayersParser from "geostyler-openlayers-parser";
import GeoJSONParser from "geostyler-geojson-parser";

import isElementInViewport from "./viewportHelper";

import "./App.css";
import "ol/ol.css";
import "antd/dist/antd.css";
import "./Workshop.css";
import Attributions from "./Attributions";
import { getDefaultStyle, getBaseLayer, getCovidLayer } from

import { MapComponent } from "@terrestris/react-geo";

import { Style as GsStyle } from "geostyler";

import covidDeath from "./data/covid-death.json";

const defaultOlStyle = getDefaultStyle();

var base = getBaseLayer();
var vector = getCovidLayer(covidDeath);

const center = [0, 8000000];

const map = new OlMap({
  view: new OlView({
    center: center,
    zoom: 2,
    projection: "EPSG:3857"
  }),
  layers: [base, vector],
  interactions: [new DragPan()]
});

const olParser = new OpenLayersParser();
const geojsonParser = new GeoJSONParser();

function App() {
  let [styles, setStyles] = useState([]);
  let [drawerVisible, setDrawerVisible] = useState(false);
```

```
let [visibleBox, setVisibleBox] = useState(0);
let [data, setData] = useState();

useEffect(() => {
  // on page init parse default style once
  // and setup the styles array
  olParser
    .readStyle(defaultOlStyle)
    .then(gsStyle => {
      const newStyles = [];
      for (var i = 0; i < 3; i++) {
        newStyles.push(JSON.parse(JSON.stringify(gsStyle))
      }
      setStyles(newStyles);
    })
    .catch(error => console.log(error));
}, []);

useEffect(() => {
  // parse data as soon as it changes
  geojsonParser
    .readData(covidDeath)
    .then(gsData => {
      setData(gsData);
    })
    .catch(error => console.log(error));
}, [data]);

useEffect(() => {
  // update the map layer when either visibleBox or styles
  var newStyle = styles[visibleBox];
  if (newStyle) {
    olParser
      .writeStyle(newStyle)
      .then(olStyle => {
        vector.setStyle(olStyle);
      })
      .catch(error => console.log(error));
  }
});

useEffect(() => {
  // add scroll eventlistener
  // unfortunately, this will be re-run as soon as visible
  // box changes. Otherwise we don't have visible box in o
  const getVisibleBox = () => {
    const boxes = [
      document.getElementById("ws-overlay-1"),
```

```
        document.getElementById("ws-overlay-2"),
        document.getElementById("ws-overlay-3")
      ];
      const boxIdx = boxes.findIndex(box => isElementInViewp
      return boxIdx >= 0 ? boxIdx : visibleBox;
    };

    const handleScroll = () => {
      const newVisibleBox = getVisibleBox();
      if (newVisibleBox !== visibleBox) {
        setVisibleBox(newVisibleBox);
      }
    };

    document.addEventListener("scroll", handleScroll);

    handleScroll();

    return () => {
      document.removeEventListener("scroll", handleScroll);
    };
  }, [visibleBox]);

  return (
    <div className='App'>
      <Button
        className='ws-toggle-editor-btn'
        type='primary'
        onClick={() => {
          setDrawerVisible(currentState => !currentState);
        }}
      >
        Toggle Editor
      </Button>
      <MapComponent map={map} />
      <Drawer
        title='GeoStyler Editor'
        placement='top'
        closable={true}
        onClose={() => {
          setDrawerVisible(false);
        }}
        visible={drawerVisible}
        mask={false}
      >
        <GsStyle
          style={styles[visibleBox]}
          compact={true}
```

```
            data={data}
            onStyleChange={newStyle => {
              olParser
                .writeStyle(newStyle)
                .then(olStyle => {
                  vector.setStyle(olStyle);
                })
                .catch(error => console.log(error));
              setStyles(oldStyles => {
                const newStyles = JSON.parse(JSON.stringify(ol
                newStyles[visibleBox] = newStyle;
                return newStyles;
              });
            }}
          />
        </Drawer>
        <span id='ws-overlay-1' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </span>
        <div id='ws-overlay-2' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <div id='ws-overlay-3' className='ws-overlay'>
          <h1>Overlay {visibleBox + 1}</h1>
          <p>Put your info text here</p>
        </div>
        <Attributions />
      </div>
    );
}

export default App;
```

Imprint

Authoren

- Jan Suleiman (suleiman@terrestris.de)
- Dirk Mennecke (-)