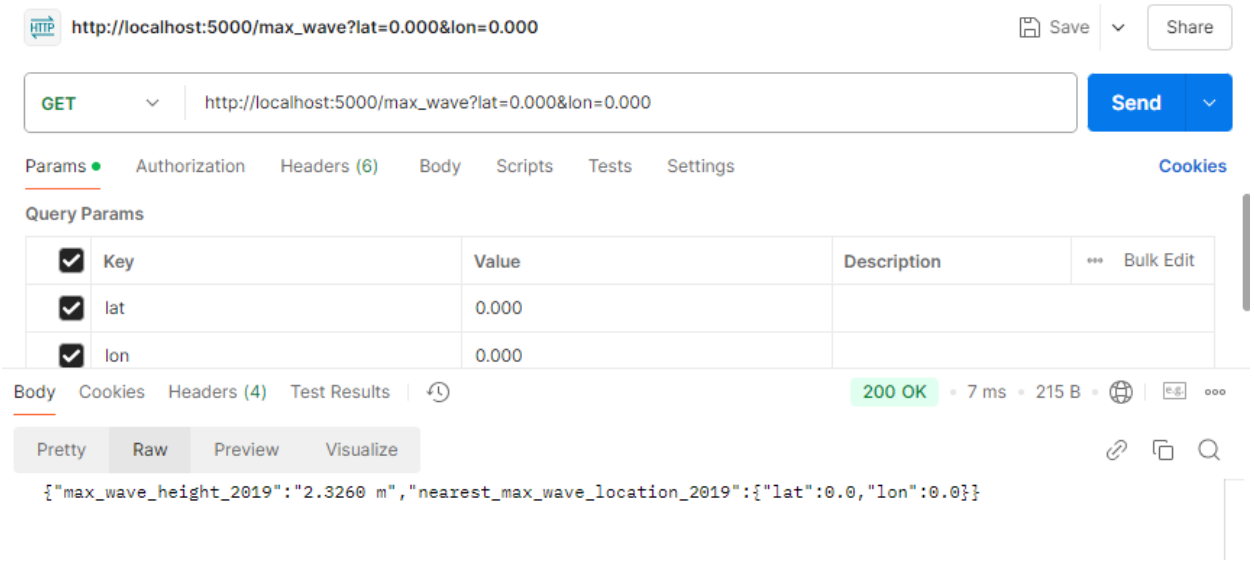
Question:1

If we try to pass latitude as 0.000, and longitude as 0.000 to our backend solution, then the max wave height we get is 2.32600m.



How we get this is by first fetching the attribute we are interested in, which is hmax in our case:

```
DATASET_PATH = "waves_2019-01-01.nc"
dataset = xr.open_dataset(DATASET_PATH)
wave_data_global = dataset["hmax"]
```

Then we select the provided latitude and longitude, but by using method=" nearest".

```
nearest_data_2019 = wave_data_global.sel(latitude=lat, longitude=lon, method="nearest")
```

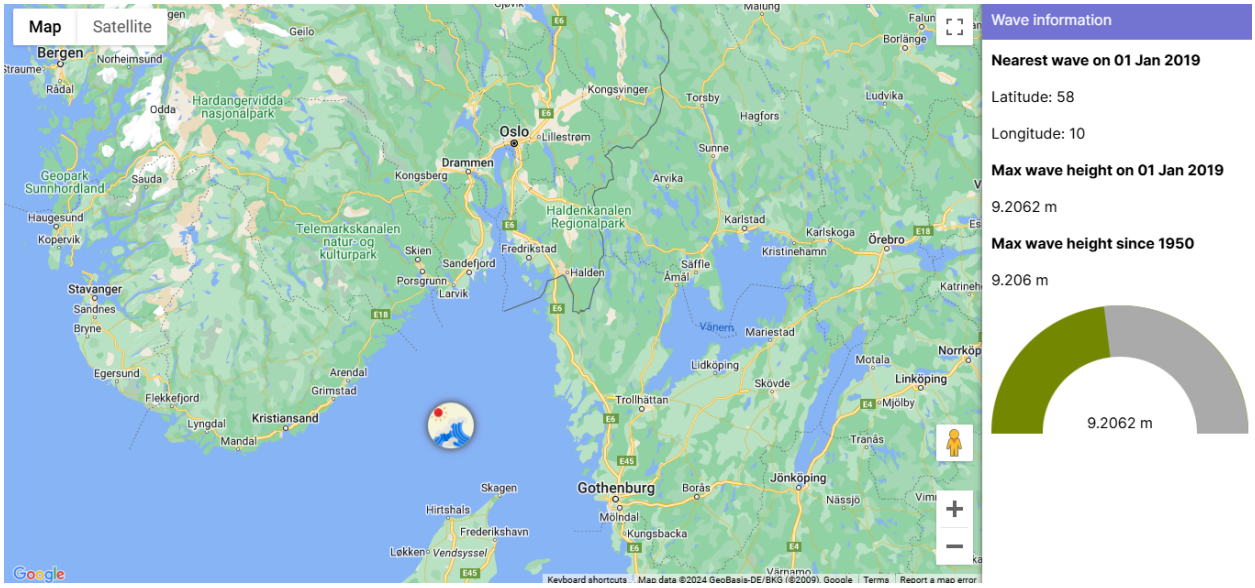
It is important as we can get the wave data not exactly in the selected location, but in the nearest location. Now that we get the wave_data in the selected location, we will fetch the max value for the selected location.

```
max_wave_2019 = nearest_data_2019.max(skipna=True).values
```

This will give the max value for the hmax in the desired location. By skipna=True we ensure to miss Nan values.

Question2:

This is how map looks like on frontend, with a sidebar,



FrontEnd

FrontEnd has been built on React, for reactivity, it as two components

1) SideBar
2) Map

We used Google Maps official library in React to implement this. We used context API to maintain the state of latitude and longitude, and max_wave, this will make the marker move based on the backend result we get where the wave is max across the selected region.
Marker is a state we have kept in our context so that it can be shared across the components of sidebar and map in this fashion.

```
  const { marker, actions } = useContext(DataContext);
```

Inside Map this state is used like this:

```
<Map
      mapId={mapId}
      defaultZoom={7}
      draggableCursor="pointer"
      gestureHandling={"greedy"}
      defaultCenter={{ lat: 59, lng: 10.5 }}
      style={{ width: "100%", height: "100%" }}
      onClick={handleClick}
    >
      {marker && (
```

```
        <AdvancedMarker position={{ lat: marker.lat, lng: marker.lng }}>
          <img className="marker-icon" src={waveIcon} alt="" />
        </AdvancedMarker>
      )}    </Map>
```

Inside the Sidebar, the state marker is used like this

```
export function Sidebar() {
  const { marker } = useContext(DataContext);
  const waveAssessment = useMemo(() => {
    const heightString = marker?.waveHeight2019?.split(" ")[0];
    if (!heightString) return null;
    const height = parseFloat(heightString);
    if (isNaN(height)) return null;
    return assessWaveDanger(height);
  }, [marker]);
  return (
    <>
      <div className="header padding">Wave information</div>
      <div className="padding">
        <b>Nearest wave on 01 Jan 2019</b>
        <p>Latitude: {marker?.lat}</p>
        <p>Longitude: {marker?.lng}</p>
        <b>Max wave height on 01 Jan 2019</b>
        <p>{marker?.waveHeight2019}</p>
        <b>Max wave height since 1950</b>
        <p>{marker?.maxWaveHeight}</p>
        {waveAssessment && (···
        )}
```

Instead of using **waveAssesment** as a local state like a usual react state variable, we are assigning it with useMemo.

It is important to note, because if we would have been changing it like setWaveAssessment function inside useEffect(), then this could cause additional component rerendering,(one rerendering when marker changes, 2nd rerendering when setWaveAssessment is called) which we avoided here.

With useMemo, as soon as marker state changes the waveAssessment changes without any additional rerendering.

waveAssessment has been used to show the rate of hazardness of the max_wave result in the form of a gauge.

```
{waveAssessment && (
          <div
            className="gauge"
            style={
              {
                width: "100%",
                "--gauge-rotation": `${waveAssessment.gaugeAngle}deg`,
                "--gauge-color": waveAssessment.color,
              } as React.CSSProperties
            }
          >
            <div className="percentage"></div>
            <div className="mask"></div>
            <span className="value">{marker?.waveHeight2019}</span>
          </div>
        )}
```

The variables used –gauge-rotation, –gauge-color will be used inside css, to transform and change the color of hazard meter gauge.

The wave assessment above is using waveAssessDanger function, it will:

1) render the gauge if the received values from backend contains float, (which means if values exist), also
2) It shows the color of how hazardous the max wave data can be.
3) If we receive no value available kind of string, then it does not show gauge.

It is implemented as follows:

```
type WaveDangerAssessment = {
  gaugeAngle: number;
  title: string;
  color: string;
};

export function assessWaveDanger(waveHeight: number): WaveDangerAssessment {
  if (waveHeight < 0) {
    throw new Error("Wave height cannot be negative.");
  }

  const gaugeAngle: number = (waveHeight / 20) * 180;
```

```
  let title: string;
  if (waveHeight <= 1.5) {
    title = "Safe (Calm to Moderate)";
  } else if (waveHeight <= 2.5) {
    title = "Caution (Moderate Waves)";
  } else if (waveHeight <= 4) {
    title = "Dangerous (Rough Seas)";
  } else {
    title = "Highly Hazardous (Very Rough or Higher)";
  }

  // Generate a continuous color gradient (green to red)
  const color = getColorFromGaugeAngle(gaugeAngle);

  return { gaugeAngle, title, color };
}

function getColorFromGaugeAngle(gaugeAngle: number): string {
  // Define the start (green) and end (red) colors
  const startColor = { r: 0, g: 255, b: 0 }; // Green
  const endColor = { r: 255, g: 0, b: 0 }; // Red

  // Interpolate between the start and end colors based on the gaugeAngle
  const r = Math.round(
    startColor.r + (endColor.r - startColor.r) * (gaugeAngle / 180)
  );
  const g = Math.round(
    startColor.g + (endColor.g - startColor.g) * (gaugeAngle / 180)
  );
  const b = Math.round(
    startColor.b + (endColor.b - startColor.b) * (gaugeAngle / 180)
  );

  return `rgb(${r}, ${g}, ${b})`;
}
```

4) This utils will return the gaugeAngle, color, and title.
5) I have set some hardcoded values for assessing hazard and safe wave data thereshold . According to the returned angle, color from this function, the CSS will set the color and rotation for the gauge.

As discussed the marker is a shared state, with an associated action changing it, is implemented under Context API, Iimplement this store in the following way:

```
type Marker = {
  lat: number;
  lng: number;
  waveHeight2019: string;
  maxWaveHeight: string;
};


type Data = {
  marker?: Marker;
  actions: {
    loadData: (lat: number, lng: number) => void;
  };
};

const defaultData: Data = {
  actions: {
    loadData: () => {},
  },
};

export const DataContext = createContext(defaultData);

export const DataProvider = ({ children }: { children: React.ReactNode }) => {
  const [marker, setMarker] = useState<Marker | undefined>();
  const loadData = async (lat: number, lng: number) => {
    const response = await axios.get(`/max_wave?lat=${lat}&lon=${lng}`);
    if (response) {
      console.log(response.data);
      setMarker({
        waveHeight2019: response.data.max_wave_height_2019,
        maxWaveHeight: response.data.max_wave_height_since_1950,
        lat: response.data.nearest_max_wave_location_2019.lat,
        lng: response.data.nearest_max_wave_location_2019.lon,
      });
    }
  };
```

In the above code we can see that loadData action is calling the backend api to fetch the max_wave for a given coordinate. It will get backend result and set the value of the shared state i.e, marker.

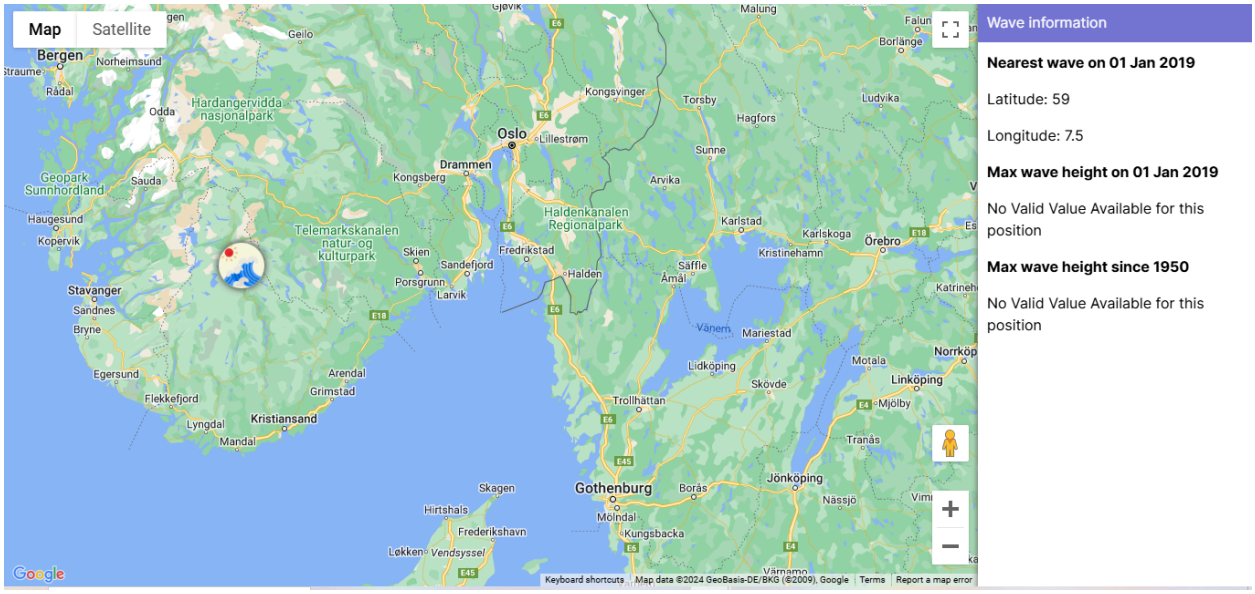
This loadData action is invoked on onclick event of the map.

```
const mapId = "somerandomid";
export function MapView() {
  const { marker, actions } = useContext(DataContext);
  const handleClick = async (ev: MapMouseEvent) => {
    if (!ev.detail.latLng) {
      console.error("Event does not have latLng.");
      alert("Invalid area selected");
      return;
    }
    actions.loadData(ev.detail.latLng.lat, ev.detail.latLng.lng);
  };
  return (
    <Map
      mapId={mapId}
      defaultZoom={7}
      draggableCursor="pointer"
      gestureHandling={"greedy"}
      defaultCenter={{ lat: 59, lng: 10.5 }}
      style={{ width: "100%", height: "100%" }}
      onClick={handleClick}
    >
      {marker && (
        <AdvancedMarker position={{ lat: marker.lat, lng: marker.lng }}>
          <img className="marker-icon" src={waveIcon} alt="" />
        </AdvancedMarker>
      )} </Map>
```

Here the action loadData is called inside onClick of the map. As we saw above, it changes the marker state, and thus our AdvanceMarker component of Map, will change its position based on the click, and the sidebar will update its content as well.

Backend:

In addition, another thing is implemented to handle the scenario if the dataset does not contain the values for the selected location, for example:



Here in the location of land with no water, there is no value in the dataset, in this case, what we get from the max function as mentioned above:

```
max_wave_2019 = nearest_data_2019.max(skipna=True).values
print("****land location result***", max_wave_2019)
```

```
****land location result**** nan
INFO:     127.0.0.1:64060 - "GET /max_wave?lat=64.4632153796058lon=18.63062768111435 HTTP/1.1" 200 OK
****land location result**** nan
```

Although skipna=True skips the NaN values, but since the selected location values are not present in the dataset then we can get NaN. For this we put a check, and give a message string instead, otherwise, if the value does exist in the dataset, then I just give the result rounded off up to 4 decimal places.

```
if math.isnan(max_wave_2019):
        max_wave_2019_return_value="No Valid Value Available for this position"
 else:
        max_wave_2019_return_value = f"{float(max_wave_2019):.4f} m"
```

For the purpose of optimization, we don't want the dataset to get the hmax every time and get the values for 2019 or 1950-2019, since they will remain constant throughout the max_wave function. Therefore, we put the following lines of code as global variables. These lines will execute once the server starts, when the request of api '/max_wave' is invoked, then the above lines will not execute, hence saving a lot of time and unnecessary repeated executions :

```
DATASET_PATH = "waves_2019-01-01.nc"
dataset = xr.open_dataset(DATASET_PATH)
```

```
wave_data_global = dataset["hmax"]
wave_data_2019=wave_data_global.sel(time="2019-01-01")
@app.get("/max_wave")
async def read_item(lat: float, lon: float, response: Response):
```

In addition, types are crucial for us while working with data arrays, therefore we used FastAPI.

Furthermore, the response we are sending is as follows:

```
        return {
            "max_wave_height_2019": max_wave_2019_return_value,
            # "max_wave_height_since_1950": max_wave_since_1950_return_value,
            "nearest_max_wave_location_2019": {
                "lat": float(nearest_data_2019['latitude'].values),
                "lon": float(nearest_data_2019['longitude'].values),
            }
        }
```

I also send the nearest_data_2019 latitude and longitude values in response, so that the user knows where in the nearest area of the selected location, the max wave is possible, this makes our Google Map Marker move, showing the user the specific location of max_wave near the clicked region.

Question3:

We know that the available data is only for 2019. To verify this, we did the following line:

```
    print(dataset["hmax"]['time.year'])
```

This gives us:

```
<xarray.DataArray 'year' (time: 24)> Size: 192B
array([2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019,
       2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019,
       2019, 2019])
```

To get the max wave since 1950, it depends upon the dataset we get. Dataset can be of 2 types:

1) A dataset containing data from 1950-01-01 to 2019-01-01:

In this case, we need to separate the max wave data for 2019, and 1950. For separating 2019-01-01 data, we do the following:

```
wave_data_global = dataset["hmax"]
wave_data_2019=wave_data_global.sel(time="2019-01-01")
```

This line will give us the data only related to 2019-01-01.

```
max_wave_2019_return_value=""
nearest_data_2019 = wave_data_2019.sel(latitude=lat, longitude=lon,
method="nearest")
max_wave_2019 = nearest_data_2019.max(skipna=True).values
if math.isnan(max_wave_2019):
  max_wave_2019_return_value="No Valid Value Available for this position"
else:
  max_wave_2019_return_value = f"{float(max_wave_2019):.4f} m"
```

The rest of the lines are similar to the above implementations.

For max_wave data since 1950, we have to fetch the wave data from the entire time dimension, since the data is already starting from 1950. Therefore we do the following:

```
wave_data_global = dataset["hmax"]
all_max_wave_global = wave_data_global.max(dim='time', skipna=True)
max_wave_data_1950= all_max_wave_global.sel(latitude=lat,longitude=lon,
method="nearest")
if math.isnan(max_wave_data_1950):
   max_wave_since_1950_return_value="No Valid Value Available for this position"
else:
   max_wave_since_1950_return_value = f"{float(max_wave_data_1950):.3f} m"
```

Here we get the max data across the time dimension, which means it will give the max wave data across all the times, and then we apply skipna=True, which skips missing values. Then we apply the latitude and longitude to filter the maximum hmax value across the desired latitude and longitude.

2) A dataset containing data from 1900-01-01(older) to 2019-01-01:

Separating 2019-01-01 data will be done in a similar way I mentioned above. For taking max wave data from 1950 to 2019, we have to filter the time range data, because we have older dates as well which we are not interested in, therefore mentioning time range is crucial while getting hmax value from dataset.

```
wave_data_global = dataset["hmax"]
wave_data_filtered=wave_data_global.sel(time=slice("1950-01-01", "2019-01-01"))
```

First we get the time range in which the data should be selected, by using slice function. Now that we have well-defined timeline data, we can get max values of wave out of them, and then apply location filters on it as usual.

```
all_max_wave_global = wave_data_filtered.max(dim='time', skipna=True)
max_wave_data_1950= all_max_wave_global.sel(latitude=lat,longitude=lon,
method="nearest")
if math.isnan(max_wave_data_1950):
  max_wave_since_1950_return_value="No Valid Value Available for this position"
else:
  max_wave_since_1950_return_value = f"{float(max_wave_data_1950):.3f} m"
```