Inhaltsverzeichnis

App.py

1. preload_status

Inputs:

- Keine direkten Eingabeparameter.

  (Die Funktion wird als Flask-Routen-Handler aufgerufen und nutzt den globalen Status preloading_complete.)

Outputs:

- JSON-Antwort:
  - Falls preloading_complete True ist: {"status": "done"}
  - Andernfalls: {"status": "loading"}

Erklärung:

Diese Funktion prüft den globalen Ladezustand der Stations- und Inventardaten (gekennzeichnet durch preloading_complete) und gibt den entsprechenden Status als JSON zurück. Verwendet wird Flask mit der jsonify-Methode, um eine standardisierte JSON-Antwort zu generieren.

Verwendete Technologien/Bibliotheken:

- Flask (für Routen und JSON-Antworten)
- Flask-CORS (zur Unterstützung von Cross-Origin-Requests)

2. index

Inputs:

- Keine direkten Eingabeparameter.

  Die Funktion wird als Flask-Routen-Handler aufgerufen.

Outputs:

- Rendert und gibt das HTML-Template index.html zurück, wobei der Parameter station_count übergeben wird.

Erklärung:

Diese Funktion ruft die Hilfsfunktion load_stations auf, um die aktuellen Stationsdaten zu laden (oder aus dem Cache zu holen), zählt die Anzahl der

geladenen Stationen und rendert anschließend das Template index.html unter Übergabe der Stationenzahl.

Verwendete Technologien/Bibliotheken:

- Flask (für Routing, Rendering von Templates mit render_template)
- Pandas (in der aufgerufenen Funktion load_stations)

3. haversine

Inputs:

- lat1 (number): Breitengrad des ersten Punkts.
- lon1 (number): Längengrad des ersten Punkts.
- lat2 (number): Breitengrad des zweiten Punkts.
- lon2 (number): Längengrad des zweiten Punkts.

Outputs:

- Rückgabewert (number): Die berechnete Entfernung zwischen den beiden Punkten in Kilometern.

Erklärung:

Diese Funktion berechnet die Luftlinienentfernung zwischen zwei geographischen Punkten auf der Erde mithilfe der Haversine-Formel. Die Umrechnung der Grad in Bogenmaß erfolgt mit der Funktion radians aus dem math-Modul. Anschließend werden die trigonometrischen Funktionen (sin, cos, atan2) verwendet, um die Distanz zu berechnen.

Verwendete Technologien/Bibliotheken:

- math (für mathematische Funktionen wie radians, sin, cos, sqrt, atan2)

4. load_stations

Inputs:

- Keine direkten Parameter.

(Nutzen globaler Variable: cached_stations)

Outputs:

• Rückgabewert: Ein Pandas DataFrame, der die Stationsdaten enthält, ggf. gefiltert auf Stationen, die sowohl TMIN- als auch TMAX-Daten liefern.

• Nebenwirkung: Setzt den globalen Cache cached_stations.

Erklärung:

Diese Funktion lädt die Stationsdaten von NOAA, sofern sie nicht bereits im Cache (cached_stations) vorhanden sind. Dabei wird eine Textdatei von der NOAA-URL abgerufen und mithilfe von pandas.read_fwf in einen DataFrame umgewandelt. Anschließend wird geprüft, ob die zugehörigen Inventardaten vorhanden sind – wenn ja, werden nur die Stationen ausgewählt, die in der Inventarliste sowohl TMIN als auch TMAX aufweisen.

Verwendete Technologien/Bibliotheken:

• requests (zum Herunterladen der Daten)

• pandas (zur Datenverarbeitung und -manipulation)

• io.StringIO (zum Umwandeln des heruntergeladenen Texts in einen File-ähnlichen Stream)

5. load_inventory

Inputs:

• Keine direkten Parameter.

(Nutzen globaler Variable: cached_inventory)

Outputs:

• Rückgabewert: Ein Pandas DataFrame, der die Inventardaten enthält.

• Nebenwirkung: Setzt den globalen Cache cached_inventory.

Erklärung:

Diese Funktion lädt die Inventardaten von NOAA, sofern diese nicht bereits im Cache vorhanden sind. Die heruntergeladenen Daten (Textdatei) werden mittels pandas.read_fwf in einen DataFrame konvertiert. Es erfolgt eine Vorverarbeitung, z. B. das Entfernen ungültiger Einträge.

Verwendete Technologien/Bibliotheken:

• requests (zum Herunterladen der Daten)

• pandas (für die Umwandlung und Verarbeitung der Daten)

• io.StringIO

6. fetch_and_filter_stations

Inputs:

- lat (number): Breitengrad des Suchzentrums.
- lon (number): Längengrad des Suchzentrums.
- radius_km (number): Suchradius in Kilometern.

Outputs:

- Rückgabewert: Ein gefilterter Pandas DataFrame, der nur Stationen enthält, deren Entfernung vom angegebenen Punkt kleiner oder gleich dem Suchradius ist.
- Nebenwirkung: Berechnung einer zusätzlichen Spalte “DISTANCE” im DataFrame.

Erklärung:

Diese Funktion ruft zunächst alle Stationsdaten (mittels load_stations) ab. Anschließend wird für jede Station die Entfernung vom angegebenen Punkt mithilfe der haversine-Funktion berechnet. Es wird ein DataFrame zurückgegeben, in dem nur die Stationen enthalten sind, deren berechnete Distanz kleiner oder gleich dem angegebenen Radius ist – sortiert nach der Entfernung.

Verwendete Technologien/Bibliotheken:

- Pandas (zur Datenmanipulation und -filterung)
- Die zuvor definierte Funktion haversine

7. parse_ghcnd_csv_from_string

Inputs:

- data (string): CSV-Daten im Textformat, wie sie von der NOAA bereitgestellt werden.

Outputs:

- Rückgabewert: Ein Pandas DataFrame mit den Spalten “DATE”, “ELEMENT” und “VALUE”.
- Nebenwirkung: Fehlerausgabe über print, falls das Parsen fehlschlägt.

Erklärung:

Diese Funktion konvertiert einen CSV-Text (vom NOAA-Server abgerufen) in einen Pandas DataFrame. Zunächst werden die Spaltennamen und Datentypen definiert. Anschließend wird das Datum in ein Datum-Format konvertiert und numerische Werte werden in Zahlen umgewandelt. Werte, die als -9999 codiert sind, werden als fehlend interpretiert und entfernt.

Verwendete Technologien/Bibliotheken:

• pandas (zum Einlesen und Verarbeiten der CSV-Daten)

• io.StringIO (zum Umwandeln des Strings in einen file-like Stream)

8. parse_ghcnd_dly_from_string

Inputs:

• data (string): Rohdaten im .dly-Format (fixe Breiten pro Zeile), bereitgestellt von NOAA.

Outputs:

• Rückgabewert: Ein Pandas DataFrame, dass Datensätze mit den Spalten “DATE”, “ELEMENT” und “VALUE” enthält.

Erklärung:

Diese Funktion parst das .dly-Datenformat, das monatlich organisierte Datenzeilen enthält. Für jede Zeile wird der Station-ID, das Jahr, der Monat sowie der jeweilige Messwert für jeden Tag extrahiert. Es werden nur gültige Werte (nicht -9999) übernommen, und das Datum wird mittels pd.Timestamp konstruiert. Die Ergebnisse werden in einen DataFrame überführt.

Verwendete Technologien/Bibliotheken:

• pandas (zur Erstellung und Verwaltung des DataFrames)

9. fetch_weather_data

Inputs:

• station_id (string): Die ID der Wetterstation, für die Daten abgerufen werden sollen.

Outputs:

• Rückgabewert: Ein Pandas DataFrame mit den Wetterdaten (Spalten: “DATE”, “ELEMENT”, “VALUE”) oder None, falls keine Daten abgerufen werden konnten.

Erklärung:

Diese Funktion versucht, alle verfügbaren Wetterdaten für eine gegebene Station abzurufen. Zuerst wird der CSV-Datensatz aus dem by_station-Verzeichnis abgerufen. Falls der CSV-Abruf fehlschlägt, wird als Fallback versucht, die .dly-Datei aus dem Verzeichnis all zu laden. Je nach Erfolg wird der entsprechende Parser (parse_ghcnd_csv_from_string oder parse_ghcnd_dly_from_string) aufgerufen, um die Daten in einen Pandas DataFrame zu überführen.

Verwendete Technologien/Bibliotheken:

• requests (zum Abrufen der Daten von externen URLs)

• pandas (für die Datenverarbeitung)

• Die zuvor definierten Parser-Funktionen

• Flask-Logging (mittels print)

10. get_stations (Flask-Routen-Handler)

Inputs:

• URL-Parameter (Query-Parameter):

• latitude (string, wird in float konvertiert): Breitengrad des Suchpunkts.

• longitude (string, wird in float konvertiert): Längengrad des Suchpunkts.

• radius_km (string, wird in float konvertiert): Suchradius in Kilometern.

• station_count (string, optional, Default: 10, wird in int konvertiert): Maximale Anzahl der zurückzugebenden Stationen.

• start_year (string, wird in int konvertiert): Startjahr des gewünschten Zeitraums.

• end_year (string, wird in int konvertiert): Endjahr des gewünschten Zeitraums.

Outputs:

• Rückgabewert: JSON-Array mit Stationen (als Liste von Dictionaries), die die Kriterien (geographische Lage, Inventar, Zeitraum) erfüllen.

• Oder eine JSON-Fehlermeldung mit Statuscode 400 oder 404 bei ungültigen Parametern bzw. fehlenden Daten.

Erklärung:

Diese Flask-Route verarbeitet einen GET-Request, filtert Stationen anhand der übergebenen Parameter (Position, Radius, Anzahl, Zeitraum) und inventarbezogener Kriterien (Stationen, die TMIN- und TMAX-Daten besitzen). Es werden zunächst die Stationsdaten mittels fetch_and_filter_stations geladen und anschließend mithilfe der Inventardaten (über load_inventory) weiter gefiltert. Das Ergebnis wird als JSON zurückgegeben.

Verwendete Technologien/Bibliotheken:

• Flask (Routing, Request-Parameter, JSON-Antworten mittels jsonify)

• Pandas (für Datenfilterung und -verarbeitung)

• requests (indirekt, da in den Hilfsfunktionen verwendet)

11. get_weather_data (Flask-Routen-Handler)

Inputs:

• URL-Parameter (Query-Parameter):

• station_id (string): Die Kennung der Station, für die Wetterdaten abgerufen werden sollen.

• start_year (string): Das Startjahr des gewünschten Datenzeitraums.

• end_year (string): Das Endjahr des gewünschten Datenzeitraums.

Outputs:

• Rückgabewert: JSON-Array der Wetterdaten, die im angegebenen Zeitraum liegen.

• Oder eine JSON-Fehlermeldung mit Statuscode 400 oder 404, falls keine Station-ID angegeben wurde oder keine Daten gefunden wurden.

Erklärung:

Diese Funktion ruft die Wetterdaten für eine bestimmte Station ab, indem sie die Funktion fetch_weather_data verwendet. Anschließend wird der Gesamtzeitraum der verfügbaren Daten geloggt und die Daten lokal (mittels

Pandas) anhand des übergebenen Jahresbereichs gefiltert. Das gefilterte Ergebnis wird dann als JSON-Antwort zurückgegeben.

Verwendete Technologien/Bibliotheken:

- Flask (für Routing, Abrufen der Query-Parameter, JSON-Antworten)
- Pandas (für die Datenfilterung)
- requests (indirekt, da in fetch_weather_data verwendet wird)

12. handle_global_error

Inputs:

- error (Exception): Die aufgetretene Ausnahme, die global abgefangen wird.

Outputs:

- Rückgabewert: JSON-Fehlermeldung mit dem Fehlertext und einem HTTP-Statuscode 500.

Erklärung:

Diese Funktion dient als globaler Fehler-Handler für die Flask-Applikation. Sie fängt alle nicht weiter behandelten Ausnahmen ab, loggt den Fehler über print und gibt eine standardisierte JSON-Antwort zurück, die den Fehler sowie Details enthält.

Verwendete Technologien/Bibliotheken:

- Flask (Error-Handling, @app.errorhandler Dekorator, jsonify)

13. error (nur im Testmodus verfügbar)

Inputs:

- Keine direkten Eingabeparameter.

(Wird nur aufgerufen, wenn die App im Testmodus läuft.)

Outputs:

- Löst absichtlich eine Exception aus.

Erklärung:

Diese Route wird nur im Testmodus aktiviert (app.config.get("TESTING")) und dient dazu, absichtlich einen Fehler zu erzeugen, um die Funktion des globalen Fehler-Handlers zu überprüfen.

Verwendete Technologien/Bibliotheken:

- Flask (Routing)
- Python Exception Handling

14. background_load (Interne Funktion im Haupt-Block)

Inputs:

- Keine direkten Eingabeparameter.

(Greift auf globale Variablen zu: preloading_complete)

Outputs:

- Lädt einmalig die Stations- und Inventardaten (durch Aufruf von load_stations und load_inventory) und setzt anschließend die globale Variable preloading_complete auf True.

Erklärung:

Diese Funktion wird in einem separaten Hintergrund-Thread ausgeführt, sobald die Applikation startet. Sie sorgt dafür, dass die notwendigen Daten (Stationen und Inventardaten) im Cache vorab geladen werden, sodass beim ersten Zugriff keine Verzögerungen durch das Datenladen entstehen.

Verwendete Technologien/Bibliotheken:

- threading (zum Starten eines Hintergrund-Threads)
- Die zuvor definierten Funktionen load_stations und load_inventory

15. Hauptprogramm (if name == “main”)

Inputs:

- Keine direkten Eingabeparameter.

(Wird beim direkten Ausführen des Skripts aufgerufen.)

Outputs:

• Startet den Hintergrund-Thread zur Vorladung der Daten und startet anschließend den Flask-Webserver im Debug-Modus.

Erklärung:

Im Hauptprogramm wird zunächst der Hintergrund-Thread gestartet, der die Funktion background_load ausführt. Anschließend wird die Flask-Applikation mit aktivierten Debug-Informationen gestartet, sodass der Webserver Requests entgegennehmen kann.

Verwendete Technologien/Bibliotheken:

• Flask (zum Starten der Webapplikation mit app.run)

• Python threading (zum parallelen Laden der Daten)

Script.js

1. getInputValue

Inputs:

• id (string): Die ID des HTML-Elements, von dem der Wert ausgelesen werden soll

Outputs:

• Rückgabewert (string): Der getrimmte Wert des Elements (Leerzeichen am Anfang und Ende werden entfernt).

Erklärung:

Diese Funktion liest den Wert eines HTML-Eingabefelds anhand seiner ID aus, entfernt führende und nachfolgende Leerzeichen und gibt den bereinigten String zurück. Sie verwendet grundlegende DOM-Methoden (z. B. document.getElementById).

2. updateMapMarker

Inputs:

• Keine direkten Parameter; ruft intern die Funktion getInputValue für die Felder “latitude” und “longitude” auf.

Outputs:

- Nebenwirkung:
    - Aktualisiert den Marker auf der Leaflet-Karte, entweder durch Verschieben eines existierenden Markers oder durch Erzeugen eines neuen Markers.
    - Setzt den Kartenfokus auf die neuen Koordinaten.
    - Aktualisiert die globale Variable latitude_positive (Boolean).

Erklärung:

Die Funktion liest die aktuellen Koordinatenwerte aus den Eingabefeldern, wandelt sie in Fließkommazahlen um und prüft auf Gültigkeit. Anschließend wird ein Marker auf der Leaflet-Karte aktualisiert oder neu erstellt, falls noch keiner existiert. Die Karte wird anschließend auf die neuen Koordinaten zentriert. Die Variable latitude_positive wird gesetzt, um anzuzeigen, ob der Breitengrad positiv ist.

Verwendete Technologien/Bibliotheken:

- Leaflet (Objekte: L.marker, map.setView)
- DOM-Methoden (z. B. document.getElementById)
- JavaScript (parseFloat, isNaN)

3. updateRadiusCircle

Inputs:

- Keine direkten Parameter; verwendet die Werte aus den Eingabefeldern “latitude”, “longitude” und “radius-input” via getInputValue.

Outputs:

Nebenwirkung:

- Entfernt den alten Radius-Kreis (falls vorhanden) von der Leaflet-Karte.
- Zeichnet einen neuen Kreis (Radius in km, umgerechnet in Meter) an der angegebenen Position.
- Zentriert die Karte auf die neuen Koordinaten.

Erklärung:

Diese Funktion liest die Koordinaten und den Suchradius aus den Eingabefeldern, wandelt sie in Zahlen um und prüft diese auf Gültigkeit. Anschließend wird ein vorhandener Radius-Kreis (falls existierend) entfernt

und ein neuer Kreis mit der angegebenen Farbe, Füllfarbe und Transparenz auf der Leaflet-Karte gezeichnet. Karte wird zentriert.

Verwendete Technologien/Bibliotheken:

- Leaflet (Methoden: L.circle, map.removeLayer, map.setView)
- DOM-Methoden und JavaScript (parseFloat, console.warn)

4. selectStation

Inputs:

- row (DOM-Element, z. B. <tr>): Die Tabellenzeile, die die ausgewählte Station repräsentiert.

Outputs:

Nebenwirkungen:

- Zeigt einen Ladeindikator (via showLoading).
- Markiert die ausgewählte Zeile in der Stationstabelle.
- Setzt den Kartenfokus auf die Koordinaten der Station.
- Aktualisiert Überschriften für jährliche, saisonale und allgemeine Wetterdaten.
- Ruft asynchron fetchWeatherData auf, um Wetterdaten der Station abzurufen, und blendet anschließend den Ladeindikator aus (via hideLoading).

Erklärung:

Beim Klick auf eine Tabellenzeile wird diese Funktion aufgerufen, um die ausgewählte Station zu verarbeiten. Sie entfernt die “selected”-Klasse von allen Zeilen, fügt sie der angeklickten Zeile hinzu und extrahiert relevante Daten (ID, Latitude, Longitude, Name). Die Karte wird auf die Stationskoordinaten gesetzt, und Überschriften in der Benutzeroberfläche werden entsprechend aktualisiert. Anschließend werden Wetterdaten über einen asynchronen Aufruf geladen.

Verwendete Technologien/Bibliotheken:

- DOM-Methoden (z. B. document.querySelectorAll, row.cells)
- Leaflet (zum Zentrieren der Karte)

• Asynchrone JavaScript-Methoden (Promise, fetch)

• Eigene Hilfsfunktionen: showLoading, hideLoading, fetchWeatherData

5. toggleDropdown

Inputs:

• event (Event-Objekt): Das auslösende Ereignis, das unter anderem das Ziel-Element (event.target) enthält.

Outputs:

Nebenwirkung:

• Wechselt die Sichtbarkeit des Dropdown-Menüs, das durch das data-target Attribut des Event-Ziels identifiziert wird.

• Schließt alle anderen Dropdown-Menüs.

Erklärung:

Diese Funktion wird beim Klicken auf ein Dropdown-Symbol ausgeführt. Sie ermittelt anhand des data-target Attributs das zugehörige Dropdown-Menü und schaltet dessen Anzeige um. Gleichzeitig werden alle anderen Dropdown-Menüs geschlossen.

Verwendete Technologien/Bibliotheken:

• DOM-Methoden (z. B. document.getElementById, document.querySelectorAll)

• JavaScript (Event-Handling)

6. filterStationsByWeatherData (asynchron)

Inputs:

• stations (Array von Objekten): Eine Liste von Stations-Daten.

• startYear (string): Startjahr für den Datenzeitraum.

• endYear (string): Endjahr für den Datenzeitraum.

Outputs:

• Rückgabewert: Ein Array der Stationen, die über gültige Wetterdaten (TMIN oder TMAX) im angegebenen Zeitraum verfügen.

Erklärung:

Diese asynchrone Funktion iteriert über ein Array von Stationen und führt für jede Station eine Abfrage der Wetterdaten (über einen Fetch-Request an /get_weather_data) durch. Sie prüft, ob in den zurückgegebenen Daten mindestens ein Datensatz für TMIN oder TMAX enthalten ist. Stationen, die diese Bedingung erfüllen, werden in das Ergebnis-Array aufgenommen.

Verwendete Technologien/Bibliotheken:

• JavaScript: Asynchrone Programmierung (async/await, Promise, fetch)

7. fetchStationData (asynchron)

Inputs:

• Keine direkten Parameter; liest jedoch mehrere Werte aus Eingabefeldern: “station-count-input”, “radius-input”, “latitude”, “longitude”, “start-year” und “end-year”.

Outputs:

Nebenwirkungen:

• Führt eine Fetch-Anfrage an den Server (URL /get_stations) durch, um Stationsdaten abzurufen.

• Aktualisiert die Stationstabelle (via updateStationTable) und die Kartenmarker (via updateStationMarkers).

Erklärung:

Diese asynchrone Funktion sammelt alle erforderlichen Suchkriterien aus den Eingabefeldern und validiert diese. Anschließend wird eine URL mit Query-Parametern konstruiert, um eine Serveranfrage zu stellen. Nach erfolgreichem Abruf der Daten werden diese geparst und anschließend werden die Stationstabelle und die Kartenmarker aktualisiert.

Verwendete Technologien/Bibliotheken:

• JavaScript: Asynchrone Programmierung (async/await, fetch)

• DOM-Methoden (zum Auslesen der Eingabefelder)

• JSON-Verarbeitung

8. updateStationTable

Inputs:

• stationData (Array von Objekten): Eine Liste von Stations-Daten, die in der Tabelle dargestellt werden sollen.

Outputs:

Nebenwirkung:

• Aktualisiert den HTML-Inhalt (innerHTML) des Elements mit der ID “station-table” und füllt es mit Tabelleneinträgen, die die Stationsinformationen anzeigen.

Erklärung:

Diese Funktion leert zunächst den Inhalt des Tabellenkörpers (station-table) und prüft, ob Stationsdaten vorhanden sind. Falls nicht, wird eine entsprechende Nachricht angezeigt. Für jede Station wird eine Tabellenzeile generiert, die die Station-ID, Breitengrad, Längengrad und den Namen enthält. Beim Klick auf eine Zeile wird die Funktion selectStation aufgerufen.

Verwendete Technologien/Bibliotheken:

• DOM-Methoden (z. B. document.getElementById, document.createElement, element.innerHTML)

• JavaScript (Array-Iteration, Event-Listener)

9. updateStationMarkers

Inputs:

• stationData (Array von Objekten): Eine Liste von Stations-Daten, die als Marker auf der Karte angezeigt werden sollen.

Outputs:

Nebenwirkung:

• Entfernt alle vorhandenen Marker von der Karte und fügt neue Marker für jede Station hinzu.

• Aktualisiert das globale Array stationMarkers.

Erklärung:

Diese Funktion entfernt zunächst alle existierenden Marker von der Leaflet-Karte, leert das Marker-Array und fügt für jede Station aus den übergebenen Daten einen neuen Marker hinzu. Jeder Marker wird so konfiguriert, dass beim Klick die Funktion selectStationByData aufgerufen wird, um weitere Details anzuzeigen.

Verwendete Technologien/Bibliotheken:

• Leaflet (zum Erstellen von Markern mit L.marker, Hinzufügen mit addTo(map))

• DOM-Methoden und JavaScript (Array-Iteration, console.log)

10. selectStationByData

Inputs:

• station (Objekt): Ein Objekt, das die Stationsdaten enthält (z. B. ID, LATITUDE, LONGITUDE, NAME).

Outputs:

Nebenwirkung:

• Zentriert die Karte auf die Station.

• Aktualisiert Überschriften für jährliche, saisonale und Wetterdatenanzeigen.

• Löst das Laden der Wetterdaten durch den Aufruf von fetchWeatherData aus.

Erklärung:

Diese Funktion wird aufgerufen, wenn ein Kartenmarker ausgewählt wird. Sie liest die Koordinaten und den Namen der Station aus, zentriert die Leaflet-Karte darauf und aktualisiert diverse Überschriften in der Benutzeroberfläche. Anschließend werden Wetterdaten für die Station geladen.

Verwendete Technologien/Bibliotheken:

• Leaflet (zum Zentrieren der Karte mit map.setView)

• DOM-Methoden (z. B. document.querySelector)

• Asynchrone Funktionen (Aufruf von fetchWeatherData)

11. fetchWeatherData (asynchron)

Inputs:

- stationId (string): Die Kennung der Wetterstation.
- startYear (string): Das Startjahr des gewünschten Datenzeitraums.
- endYear (string): Das Endjahr des gewünschten Datenzeitraums.

Outputs:

Nebenwirkungen:

- Führt eine Fetch-Anfrage an den Server durch (URL /get_weather_data), um Wetterdaten abzurufen.
- Bei Erfolg wird die Funktion processWeatherData mit den abgerufenen Daten aufgerufen.
- Im Fehlerfall wird eine Fehlermeldung ausgegeben und ggf. im DOM angezeigt.

Erklärung:

Diese asynchrone Funktion baut eine URL mit den erforderlichen Query-Parametern zur Abfrage von Wetterdaten auf und sendet einen HTTP-Request an den Server. Nach erfolgreicher Antwort werden die Rohdaten geparst und an processWeatherData übergeben. Bei einem Fehler wird eine Fehlermeldung in der Konsole und gegebenenfalls im Chart-Bereich angezeigt.

Verwendete Technologien/Bibliotheken:

- JavaScript: Asynchrone Programmierung (async/await, fetch)
- DOM-Methoden (z. B. document.getElementById)

12. processWeatherData

Inputs:

- data (Array von Objekten): Rohdaten der Wetterstation, enthalten u. a. Datum, ELEMENT und VALUE.

Outputs:

Nebenwirkungen:

• Transformiert und normalisiert die Wetterdaten (z. B. Umwandlung des Datums, Skalierung des Wertes).

• Aggregiert jährliche Durchschnittswerte (TMIN, TMAX) und saisonale Werte mittels D3.js.

• Speichert die verarbeiteten Daten in window.currentWeatherDataset.

• Ruft anschließend drawChart zur Visualisierung der Daten auf.

Erklärung:

Die Funktion iteriert über die Rohdaten, konvertiert Datum und numerische Werte und skaliert die Messwerte. Anschließend werden mithilfe von D3.js Methoden (wie d3.nest, d3.mean) aggregierte jährliche und saisonale Durchschnittswerte für TMIN und TMAX berechnet. Zwei Hilfsfunktionen (getSeason und getSeasonYear) bestimmen dabei die Saisons basierend auf den Daten und der globalen Variable latitude_positive. Die aggregierten Daten werden in einem Objekt gespeichert und anschließend an drawChart übergeben, um die Visualisierung zu erstellen.

Verwendete Technologien/Bibliotheken:

• D3.js (Methoden: d3.nest, d3.mean)

• JavaScript (Array-Methoden, Datumskonvertierung)

• DOM-Methoden (für eventuelle Fehlermeldungen)

13. fillMissingYears

Inputs:

• data (Array von Objekten): Array mit Objekten, die jeweils year (number) und value (number) enthalten.

• minYear (number): Das früheste Jahr im gewünschten Bereich.

• maxYear (number): Das späteste Jahr im gewünschten Bereich.

Outputs:

• Rückgabewert (Array): Ein Array, in dem für jedes Jahr zwischen minYear und maxYear ein Objekt {year, value} enthalten ist. Falls für ein Jahr kein Wert existiert, wird value als null gesetzt.

Erklärung:

Diese Funktion erstellt eine Lücken-freie Darstellung der Jahresdaten. Sie baut eine Map aus den vorhandenen Daten, iteriert dann über den kompletten Jahrbereich und fügt für fehlende Jahre einen Eintrag mit value: null hinzu.

14. drawChart

Inputs:

• dataset (Objekt): Ein Objekt, das aggregierte Wetterdaten enthält, unter anderem:

• annualTmin (Array)

• annualTmax (Array)

• seasonalTmin (Array)

• seasonalTmax (Array)

Outputs:

Nebenwirkungen:

• Entfernt vorhandene SVG-Elemente aus dem DOM.

• Erzeugt eine neue SVG-Grafik, zeichnet Achsen, Linien und eine Legende.

• Bindet Maus-Interaktionen (Tooltip, Fokuslinien) an die Grafik.

• Speichert die aktuell gezeichneten Linien in window.currentLines.

Erklärung:

Diese Funktion visualisiert die aggregierten Wetterdaten mithilfe von D3.js. Zunächst wird der vorhandene Inhalt in den Ziel-DOM-Elementen (z. B. “#d3-chart” und “#chart-legend”) entfernt. Es werden Skalierungen für die x- und y-Achse basierend auf den Daten berechnet. Anschließend wird eine SVG-Fläche erstellt, in der Linien (mit Hilfe von d3.line) für jährliche und saisonale Durchschnittswerte gezeichnet werden. Eine Legende ermöglicht das Ein- und Ausblenden einzelner Datenreihen, und ein Tooltip sowie ein Fokuselement (Focus-Circles) liefern interaktive Informationen beim Überfahren mit der Maus.

Verwendete Technologien/Bibliotheken:

• D3.js (SVG-Erstellung, Skalierung, Achsen, Linien, Interaktionen, d3.nest, d3.line, d3.axis)

15. drawDataTable

Inputs:

• dataset (Objekt): Enthält aggregierte Wetterdaten, z. B. annualTmin, annualTmax, seasonalTmin, seasonalTmax.

• lines (Array von Objekten): Array der aktuell angezeigten Datenreihen

Outputs:

Nebenwirkung:

• Aktualisiert den HTML-Inhalt der Container mit den IDs “annual-data-table” und “seasonal-data-table” und füllt sie mit tabellarisch aufbereiteten Daten.

Erklärung:

Die Funktion filtert zunächst die sichtbaren Datenreihen aus dem übergebenen lines-Array und erstellt dann zwei separate Tabellen: eine für jährliche Durchschnittswerte und eine für saisonale Werte. Die Daten werden sortiert und in HTML-Tabellenform gebracht, die anschließend in den entsprechenden Container-Elementen angezeigt werden.

16. confirmSelection (asynchron)

Inputs:

• Keine direkten Parameter; ruft intern andere Funktionen auf.

Outputs:

Nebenwirkungen:

• Zeigt einen Ladeindikator (via showLoading).

• Speichert die aktuellen Suchkriterien (via saveSearchCriteria).

• Ruft fetchStationData auf, um Stationsdaten abzurufen.

• Blendet den Ladeindikator wieder aus (via hideLoading), unabhängig vom Ergebnis.

Erklärung:

Diese asynchrone Funktion wird ausgelöst, wenn der Benutzer die Suche bestätigt. Sie zeigt zunächst einen Ladeindikator, speichert dann die eingegebenen Suchkriterien und ruft anschließend fetchStationData auf, um

die relevanten Stationsdaten vom Server zu laden. Abschließend wird der Ladeindikator ausgeblendet.

Verwendete Technologien/Bibliotheken:

• JavaScript: Asynchrone Programmierung (async/await)

• DOM-Methoden (für Anzeige und Speicherung von Werten)

• Eigene Hilfsfunktionen: showLoading, hideLoading, saveSearchCriteria

17. saveSearchCriteria

Inputs:

• Keine direkten Parameter; liest Werte aus den Eingabefeldern “station-count-input”, “radius-input”, “start-year”, “end-year”, “latitude” und “longitude” via getInputValue.

Outputs:

Nebenwirkung:

• Loggt die aktuellen Suchkriterien als JSON-String in die Konsole.

Erklärung:

Diese Funktion sammelt alle Suchkriterien aus den entsprechenden Eingabefeldern und erstellt ein Objekt, das diese Werte enthält. Anschließend wird das Objekt als JSON-String formatiert und in der Konsole ausgegeben.

Verwendete Technologien/Bibliotheken:

• JavaScript (JSON.stringify, console.log)

18. setStationCount

Inputs:

• value (number oder string): Der neue Wert für die maximale Anzahl der anzuzeigenden Stationen.

Outputs:

Nebenwirkung:

• Setzt den Wert des Eingabefeldes mit der ID “station-count-input”.

• Blendet das Dropdown-Menü (“station-count-dropdown”) aus.

Erklärung:

Diese Funktion wird beim Auswählen eines Werts aus dem Dropdown-Menü für die Anzahl der Stationen aufgerufen. Sie aktualisiert das Eingabefeld entsprechend und schließt das Dropdown-Menü.

19. setRadiusValue

Inputs:

• value (number oder string): Der neue Wert für den Suchradius in Kilometern.

Outputs:

Nebenwirkung:

• Aktualisiert den Wert des Eingabefeldes “radius-input”.

• Blendet das zugehörige Dropdown-Menü (“radius-dropdown”) aus.

• Ruft updateRadiusCircle auf, um den neuen Radius auf der Karte darzustellen.

Erklärung:

Diese Funktion wird beim Auswählen eines Radiuswerts aus dem Dropdown-Menü ausgeführt. Sie aktualisiert das Eingabefeld, schließt das Dropdown und sorgt dafür, dass der neue Radius-Kreis auf der Karte gezeichnet wird.

Verwendete Technologien/Bibliotheken:

• Eigene Funktion: updateRadiusCircle

20. showLoading

Inputs:

• Keine direkten Parameter.

Outputs:

Nebenwirkung:

• Setzt die CSS-Display-Eigenschaft des Elements mit der ID “loading-overlay” auf “flex”, sodass ein Ladeindikator sichtbar wird.

Erklärung:

Diese Funktion zeigt den Ladeüberlagerungsbereich an, indem sie dessen CSS-Display-Wert ändert. Dies signalisiert dem Benutzer, dass eine Operation im Hintergrund läuft.

Verwendete Technologien/Bibliotheken:

- JavaScript (direkte Manipulation von Style-Eigenschaften)

21. hideLoading

Inputs:

- Keine direkten Parameter.

Outputs:

Nebenwirkung:

- Setzt die CSS-Display-Eigenschaft des Elements mit der ID “loading-overlay” auf “none”, wodurch der Ladeindikator ausgeblendet wird.

Erklärung:

Diese Funktion blendet den Ladeüberlagerungsbereich aus, indem sie dessen CSS-Display-Wert auf “none” setzt.

Verwendete Technologien/Bibliotheken:

- DOM-Methoden (z. B. document.getElementById)
- JavaScript (direkte Style-Manipulation)

22. checkPreloadStatus

Inputs:

- Keine direkten Parameter.

Outputs:

Nebenwirkung:

-

Erklärung:

Diese Funktion prüft periodisch den Ladezustand der vorab geladenen Daten, indem sie eine GET-Anfrage an /preload_status sendet. Falls die Antwort anzeigt, dass das

Vorladen abgeschlossen ist, wird der Ladeindikator ausgeblendet. Andernfalls wird die Überprüfung nach einer kurzen Verzögerung fortgesetzt.

Verwendete Technologien/Bibliotheken:

- JavaScript: Fetch API (für HTTP-Anfragen), Promises, setTimeout

Index.html

Dokumenttyp und Sprache:

Das Dokument beginnt mit der Definition des HTML5-Dokuments und legt die Sprache auf Englisch fest, was für Browser und Suchmaschinen relevant ist.

Kopfbereich (Head)

Im Head werden wichtige Meta-Informationen definiert. Es wird die Zeichenkodierung auf UTF-8 gesetzt und ein responsives Viewport-Management eingerichtet, um eine optimale Darstellung auf verschiedenen Geräten zu gewährleisten. Der Seitentitel wird definiert, sodass er in der Browser-Registerkarte erscheint. Zudem werden externe Stylesheets eingebunden: Zum einen die CSS-Datei der Leaflet-Bibliothek für die Kartenanzeige und zum anderen eine eigene CSS-Datei, deren Pfad durch Flask dynamisch generiert wird.

Body-Bereich – Struktur und Inhalte

Der Body enthält mehrere Bereiche, die jeweils spezifische Aufgaben erfüllen:

- Loading-Overlay:
  Dieser Abschnitt zeigt ein Lade-Overlay mit einer Wrapper-Struktur und einer Reihe von Elementen zur Animation an. Es dient dazu, dem Nutzer während des Ladevorgangs visuelles Feedback zu geben.
- Grid-Layout:

Der Hauptinhalt ist in einem Rasterlayout organisiert, das verschiedene Container enthält:

- Obere Container:
  Vier separate Container sind für die Anzeige von Informationen und interaktive Eingaben zuständig. Dabei wird dynamisch die Anzahl verfügbarer Stationen angezeigt, der Nutzer kann über Dropdown-Menüs die Anzahl der Stationen sowie den Radius in Kilometern auswählen. Zusätzlich gibt es Eingabefelder zur Definition eines Zeitraums (Start- und Endjahr) mit vorgegebenen Minimal- und Maximalwerten.
- Kartenanzeige:
  Ein großer Container ist ausschließlich für die Darstellung einer interaktiven Karte vorgesehen. Die Einbindung der Leaflet-Bibliothek ermöglicht dabei die Initialisierung und Verwaltung der Karte.
- Rechte Container:
  Auf der rechten Seite finden sich zwei Bereiche: Einer erlaubt die Eingabe von Koordinaten (Breiten- und Längengrad) und bietet einen Bestätigungsbutton, um Aktionen (wie das Zentrieren der Karte) auszulösen. Der andere Bereich zeigt eine Tabelle, in der die Stationen

mit Details wie ID, Koordinaten und Name dynamisch dargestellt werden.

- Datenvisualisierung:
  Es existieren separate Container für Diagramme und Tabellen, die mithilfe von D3.js und eigenen JavaScript-Funktionen erstellt werden. Hier werden zum Beispiel jährliche und saisonale Durchschnittswerte visualisiert.

Einbindung von JavaScript und css

1. Leaflet-Bibliothek:
   Diese Bibliothek wird eingebunden, um eine interaktive Kartenanzeige zu ermöglichen. Sie stellt alle notwendigen Funktionen bereit, um die Karte zu initialisieren, zu steuern und benutzerdefinierte Layer oder Marker darzustellen.
2. D3.js:
   D3.js wird eingebunden, um datenbasierte Diagramme zu erstellen. Mit dieser Bibliothek können dynamische Visualisierungen entwickelt werden, die komplexe Datenmuster und -beziehungen anschaulich darstellen.
3. Eigene JavaScript-Datei:
   Zusätzlich wird ein eigenes Skript geladen, in dem das Verhalten der interaktiven Elemente definiert wird. Dies umfasst beispielsweise die Steuerung der Dropdown-Menüs, die Initialisierung und Aktualisierung der Karte sowie die Verarbeitung und Darstellung von Daten in den Diagrammen und Tabellen.
4. styles.css
   Die CSS-Datei enthält benutzerdefinierte Stillregeln, die das Aussehen und Layout der Webanwendung bestimmen.

Style.css

1. Allgemeines Styling

Selektor:

- body

- Keine klassischen Parameter, sondern alle Eigenschaften, die hier auf den Body angewendet werden.

Erklärung:

Diese Regel definiert globale Stile für das <body>-Element.

- **Margin und Padding:** Setzt beide Werte auf 0, um unerwünschte Abstände zu vermeiden.

- **Font-Family:** Legt “Arial, sans-serif” als Schriftart fest.

- **Background:** Verwendet ein festes Hintergrundbild (hinterlegt unter ../images/Background.jpg), das zentriert und fixiert dargestellt wird.

• **Background-Size:** Das Bild wird so skaliert, dass es den gesamten Hintergrund abdeckt.

2. Box Sizing

Selektor:

• *, *::before, *::after

Outputs:

• Legt fest, dass alle Elemente und ihre Pseudoelemente das Box-Model „border-box“ verwenden.

Erklärung:

Diese Regel sorgt dafür, dass bei allen Elementen die Padding- und Border-Werte in der Gesamtbreite und -höhe berücksichtigt werden. Dies vereinfacht das Layout und verhindert unerwartete Größenänderungen.

3. BoxHeading

Selektor:

• .BoxHeading

Outputs:

• Ändert das Erscheinungsbild von Elementen mit der Klasse BoxHeading durch Anpassen der Schriftgröße und des Schriftgewichts.

Erklärung:

Diese Regel setzt die Schriftgröße auf 20px und definiert das Schriftgewicht als bold, um Überschriften in Containern hervorzuheben.

4. Container Styling

Selektor:

• .container

Outputs:

• Formatiert Container-Elemente mit einem halbtransparenten weißen Hintergrund, abgerundeten Ecken, Innenabstand (Padding) und einem leichten Schatten.

Erklärung:

Diese Regel sorgt für ein modernes Erscheinungsbild der Container:

- **Background:** Weiß mit 80 % Deckkraft (rgba(255,255,255,0.8)).
- **Border-Radius:** 15px, um die Ecken abzurunden.
- **Padding:** 15px, um Inhalte von den Rändern fernzuhalten.
- **Box-Shadow:** Ein leichter Schatten, der Tiefe verleiht.
- **Text-Align:** Zentriert den Text innerhalb des Containers.

5. Less Transparent Container

Selektor:

- .less_transparent_container

Outputs:

- Formatiert Container-Elemente mit einem halbtransparenten weißen Hintergrund, abgerundeten Ecken, Innenabstand (Padding) und einem leichten Schatten.

Erklärung:

Diese Regel sorgt für ein modernes Erscheinungsbild der Container:

- **Background:** Weiß mit 95 % Deckkraft (rgba(255,255,255,0.95)).
- **Border-Radius:** 15px, um die Ecken abzurunden.
- **Padding:** 15px, um Inhalte von den Rändern fernzuhalten.
- **Box-Shadow:** Ein leichter Schatten, der Tiefe verleiht.
- **Text-Align:** Zentriert den Text innerhalb des Containers.

6. Grid-Layout

Selektor:

- .grid-layout

Outputs:

• Ordnet enthaltene Elemente in einem Raster an, definiert 4 gleich große Spalten, legt Abstände (grid-gap) und Innenabstand (padding) fest.

Erklärung:

Diese Regel nutzt die CSS-Grid-Technologie, um ein flexibles Layout zu schaffen:

• **grid-template-columns:** Erzeugt 4 gleich breite Spalten (1fr je Spalte).

• **grid-gap:** Definiert einen Abstand von 15px zwischen den Zellen.

• **Padding:** Fügt rundum einen Abstand von 20px hinzu.

7. Buttons

Selektor:

•.button und .button:hover

Outputs:

• Stilisiert Schaltflächen mit Hintergrundfarbe, Textfarbe, abgerundeten Ecken und Hover-Effekten.

Erklärung:

Die .button-Klasse legt grundlegende Eigenschaften fest:

• **Hintergrundfarbe:** Ein Blau-Ton (rgb(54,150,213)).

• **Textfarbe:** Weiß.

• **Border:** Keine, mit abgerundeten Ecken (5px).

• **Padding:** Sorgt für ausreichend Innenabstand (8px 15px).

• **Cursor:** Zeigt beim Überfahren den Zeiger an.

• **Transition:** Weicher Übergang bei Hintergrundänderung.

Beim Hover-Effekt ändert sich die Hintergrundfarbe zu einem etwas dunkleren Blau (rgb(30,130,200)).

8. Dropdown-Menüs

Selektoren:

- .dropdown-wrapper
- .dropdown-input
- .dropdown-btn
- .dropdown-list
- .dropdown-list li
- .dropdown-list li:hover

Outputs:

- Formatierung von Dropdown-Elementen: Wrapper, Eingabefelder, Buttons und Listen.

Erklärung:

- **.dropdown-wrapper:** Definiert ein flexibles Container-Layout, zentriert Inhalte horizontal und vertikal und positioniert das Dropdown relativ.
- **.dropdown-input:** Setzt Breite, Höhe, Schriftgröße, Textausrichtung und sorgt für ausreichend Platz rechts (für den Button).
- **.dropdown-btn:** Positioniert den Button absolut im Wrapper, passt Größe und Rand an und sorgt für eine Scrollbar bei Bedarf.
- **.dropdown-list:** Positioniert die Liste direkt unter dem Eingabefeld, legt maximale Breite und Box-Sizing fest, ist standardmäßig ausgeblendet und erhält einen leichten Schatten.
- **.dropdown-list li:** Legt Polsterung und Textausrichtung für Listeneinträge fest.
- **.dropdown-list li:hover:** Ändert die Hintergrund- und Textfarbe beim Überfahren (Hover).

9. Media Queries (für kleinere Bildschirme)

Selektor:

- @media (max-width: 600px)

Outputs:

- Passt das Layout an kleinere Bildschirme an, indem es den Wrapper der Dropdown-Menüs auf 100 % Breite begrenzt.

Erklärung:

Innerhalb dieser Media Query wird die maximale Breite des Dropdown-Wrappers auf 100 % gesetzt, um eine optimale Darstellung auf kleinen Bildschirmen sicherzustellen.

Verwendete Technologien/Bibliotheken:

- CSS Media Queries

10. Karte

Selektor:

- .large-container
- #map

Outputs:

- Definiert das Layout für den Kartencontainer und die Größe sowie den Abstand der Karte.

Erklärung:

- **.large-container:** Weist den Container an, über 2 Spalten im Grid zu gehen und zentriert den Inhalt (hier die Karte).
- **#map:** Legt die Breite und Höhe der Karte fest (unter Berücksichtigung von Abständen) und fügt einen Rand (Margin) hinzu, um gleichmäßige Abstände zu gewährleisten.

Verwendete Technologien/Bibliotheken:

- CSS Grid

11. Rechte Seitencontainer

Selektor:

- .right-containers
- .small-container

Outputs:

- Ordnet rechte Container in einer Spalte an und definiert Abstände zwischen diesen sowie deren vertikale Zentrierung.

Erklärung:

• **.right-containers:** Setzt die Container als flexibles, vertikal ausgerichtetes Layout mit einem definierten Abstand (gap: 15px) und weist sie im Grid 2 Spalten zu.

• **.small-container:** Stellt sicher, dass Container in der rechten Spalte sich an ihren Inhalt anpassen und zentriert ausgerichtet sind.

Verwendete Technologien/Bibliotheken:

• CSS Flexbox, CSS Grid

12. Vollbreite Container

Selektor:

• .full-width-container

Outputs:

• Weist Container an, sich über alle 4 Spalten des Grids zu erstrecken und begrenzt gleichzeitig deren maximale Höhe.

Erklärung:

Diese Regel sorgt dafür, dass Container, die volle Breite benötigen (z. B. Diagramme oder Daten-Tabellen), im Grid über alle Spalten angezeigt werden, jedoch nicht höher als 550px werden.

Verwendete Technologien/Bibliotheken:

• CSS Grid

13. Datumseingabe

Selektor:

• .date-container

• .date-input

Outputs:

• Ordnet Datumsfelder (z. B. für Start- und Endjahr) in einer flexiblen, horizontalen Reihe an und definiert deren Größe und Aussehen.

Erklärung:

• **.date-container:** Nutzt Flexbox, um die Datumsfelder mit definierten Abständen horizontal und vertikal zu zentrieren.

• **.date-input:** Legt Breite, Padding, Textausrichtung, Rand und Schriftgröße für die Datumsfelder fest, um eine einheitliche Eingabe zu gewährleisten.

Verwendete Technologien/Bibliotheken:

• CSS Flexbox

14. Eingabefelder

Selektor:

•.input-field

Outputs:

• Definiert die Breite, das Padding, den Rand, die Schriftgröße und die zentrierte Ausrichtung für allgemeine Eingabefelder.

Erklärung:

Diese Regel stellt sicher, dass alle Eingabefelder (z. B. für Koordinaten) einheitlich aussehen und sich optisch in das Gesamtlayout einfügen.

15. Tabellen

Selektor:

• .table-container

• .large-table-container

• table

• th, td

Outputs:

• Legt die Größe, das Layout und das Aussehen von Tabellen fest, inklusive Grenzen, Abständen und Scrollverhalten.

Erklärung:

• **.table-container:** Definiert eine maximale Höhe und ermöglicht vertikales Scrollen.

• **.large-table-container:** Ähnlich wie .table-container, jedoch mit einer größeren maximalen Höhe.

• **table:** Setzt die Breite der Tabelle auf 100 % und kollabiert Rahmen.

• **th, td:** Definiert Rahmen, Padding und Textausrichtung für Kopf- und Zellen.

• **th:** Verwendet eine spezifische Hintergrund- und Textfarbe, um Tabellenüberschriften hervorzuheben.

16. Markierte Zeile

Selektor:

• .selected (sowie die Erweiterung für Tabellenzeilen, z. B. tbody tr.selected)

Outputs:

• Hebt eine ausgewählte Tabellenzeile hervor, indem Hintergrund, Textfarbe und Schriftgewicht geändert werden.

Erklärung:

Diese Regel sorgt dafür, dass beim Auswählen einer Tabellenzeile (z. B. bei Klick) diese visuell hervorgehoben wird, um dem Benutzer Feedback zu geben.

Verwendete Technologien/Bibliotheken:

• Verwendung von !important zur Priorisierung

17. Höhere Container für bessere Lesbarkeit

Selektor:

• .grid-layout > .container:nth-child(1),

• .grid-layout > .container:nth-child(2),

• .grid-layout > .container:nth-child(3),

• .grid-layout > .container:nth-child(4)

Outputs:

• Setzt eine feste Höhe (60px) und sorgt für ein zentriertes Layout dieser Container.

Erklärung:

Diese Regel stellt sicher, dass die oberen vier Container im Grid einheitlich in der Höhe sind und Inhalte (wie Überschriften) gut lesbar und zentriert dargestellt werden.

Verwendete Technologien/Bibliotheken:

• CSS-Selektoren

18. Überschriften innerhalb von Containern

Selektor:

• .grid-layout > .container h3

• .grid-layout > .container p

Outputs:

• Entfernt zusätzliche Abstände und sorgt für zentrierte Textausrichtung bei Überschriften und Absatztexten.

Erklärung:

Diese Regeln setzen Margin und Padding auf 0 und zentrieren den Text, um ein sauberes und einheitliches Erscheinungsbild in den Containern zu gewährleisten.

19. Tabellenzeilen Interaktivität

Selektor:

• tbody tr

• tbody tr:hover

• tbody tr.selected

Outputs:

• Legt den Cursor (pointer) fest, fügt einen Hover-Effekt hinzu und hebt beim Klicken die ausgewählte Zeile hervor.

Erklärung:

• **tbody tr:** Stellt den Zeiger als “pointer” ein und sorgt für eine sanfte Hintergrundänderung durch Transition.

• **tbody tr:hover:** Verändert die Hintergrundfarbe, wenn die Maus über einer Zeile schwebt.

• **tbody tr.selected:** Hebt die Zeile farblich und typografisch (fett) hervor, wenn sie ausgewählt ist.

Verwendete Technologien/Bibliotheken:

• CSS-Transitions

20. Responsives Design (Grid-Layout-Anpassung)

Selektor:

• @media (max-width: 1024px)

• @media (max-width: 768px)

Outputs:

• Passt das Grid-Layout an kleinere Bildschirmgrößen an, indem die Anzahl der Spalten reduziert wird.

Erklärung:

• Bei einer Bildschirmbreite unter 1024px wird das Grid auf 2 Spalten umgestellt.

• Bei unter 768px wird das Grid auf eine Spalte reduziert.

Dadurch wird eine bessere Darstellung auf mobilen Geräten erreicht.

Verwendete Technologien/Bibliotheken:

• CSS Media Queries

21. Chart Content

Selektor:

• #chart-content

• #d3-chart

Outputs:

• Ordnet die Container für die Diagrammdarstellung als flexibles Layout an und sorgt dafür, dass der Diagrammcontainer scrollbar ist.

Erklärung:

• **#chart-content:** Setzt ein flexibles, nicht umbrechendes Layout, in dem die Diagramme und Legenden nebeneinander angezeigt werden; bei Bedarf ist horizontales Scrollen möglich.

• **#d3-chart:** Legt eine Mindestbreite fest und stellt sicher, dass der Container seine Größe anpasst, ohne das Layout zu sprengen.

Verwendete Technologien/Bibliotheken:

• CSS Flexbox, Overflow-Einstellungen

22. Achsen-Styling

Selektor:

• .axis path

• .axis line

• .axis text

Outputs:

• Legt das Aussehen von Achsenlinien und Beschriftungen in Diagrammen fest (z. B. Farbe, Strichstärke, Schriftgröße).

Erklärung:

Diese Regeln sorgen dafür, dass die Achsen in Diagrammen (die mit D3.js erzeugt werden) ein einheitliches, graues Erscheinungsbild haben:

• **Pfad und Linien:** Graue Farbe und eine Dicke von 1.5px.

• **Text:** Graue Schrift mit 14px Größe.

Verwendete Technologien/Bibliotheken:

• speziell für D3.js erzeugte SVG-Elemente

23. Chart Legend

Selektor:

- #chart-legend
- .legend-item
- .legend-dot
- .legend-text

Outputs:

- Stellt eine Legende für Diagramme dar, mit Container, farbigen Punkten (Dots) und Beschriftungen.

Erklärung:

- **#chart-legend:** Definiert Rahmen, Padding, Schatten, minimale Breite und Flexbox-Eigenschaften, um die Legende als vertikales Menü darzustellen.
- **.legend-item:** Ordnet die einzelnen Legenden-Einträge in einer flexiblen Zeile an und ermöglicht Klick-Interaktionen.
- **.legend-dot:** Erzeugt einen kleinen, farbigen Punkt, der die jeweilige Datenreihe repräsentiert.
- **.legend-text:** Stellt den zugehörigen Text mit einem linken Abstand dar.

Verwendete Technologien/Bibliotheken:

- Flexbox

24. SVG Responsive

Selektor:

- .svg-content-responsive

Outputs:

- Erzwingt, dass eingebettete SVG-Elemente ihre Breite an den Container anpassen und dabei ihre Höhe automatisch skalieren.

Erklärung:

Diese Regel sorgt dafür, dass SVG-Grafiken, die beispielsweise durch D3.js generiert werden, flexibel und responsiv innerhalb ihres Containers dargestellt werden.

25. Loading Overlay

Selektor:

- #loading-overlay

Outputs:

• Positioniert einen über die gesamte Seite gehängten, halbtransparenten Overlay-Bereich, der einen Ladeindikator anzeigt.

Erklärung:

Diese Regel fixiert den Overlay-Bereich über dem gesamten Viewport (100vw x 100vh) und verwendet Flexbox, um den darin enthaltenen Spinner zentriert darzustellen. Der Hintergrund ist leicht transparent (grau), um den Ladezustand visuell hervorzuheben.

Verwendete Technologien/Bibliotheken:

• Flexbox, Z-Index

26. Section Wrapper

Selektor:

- section.wrapper

Outputs:

• Fügt einem Abschnitt (Wrapper) ein vertikales Padding hinzu und zentriert dessen Inhalt.

Erklärung:

Diese Regel sorgt dafür, dass Inhalte innerhalb eines <section>-Elements mit der Klasse wrapper optisch zentriert und mit ausreichend Abstand (Padding) dargestellt werden.

Verwendete Technologien/Bibliotheken:

• Flexbox

27. Keyframes – rotateRandom

Outputs:

• Erzeugt eine Animation, bei der sich Elemente (hier: Spinner) in mehreren Stufen drehen (0°, 90°, 200°, 300°, zurück zu 0°).

Erklärung:

Diese Keyframes-Definition legt den Verlauf einer Rotationsanimation fest, die zufällig anmutende Drehwinkel für einen Spinner simuliert.

Verwendete Technologien/Bibliotheken:

• Standard-CSS Animationen (@keyframes)

28. Spinner-Styles

Selektor:

• .spinner i

• .spinner i:nth-child(n) (für n = 1 bis 13)

Outputs:

• Definiert die Größe, Position, Rahmen, Farbe, und Animationseigenschaften für jedes <i>-Element innerhalb des Spinners.

Erklärung:

Diese Regeln stylen die einzelnen Kreise des Spinners:

• Jedes <i>-Element wird absolut positioniert, zentriert, erhält eine bestimmte Breite und Höhe, einen Rand (border) und wird mittels clip-path in Form gebracht.

• Durch die Verwendung der rotateRandom-Animation mit unterschiedlichen Zeiten und Richtungen (alternate / alternate-reverse) entsteht ein mehrschichtiger, animierter Ladeeffekt.

Verwendete Technologien/Bibliotheken:

• CSS Animationen, CSS Pseudoklassen (nth-child)