«can-I-go.ch» - Dokumentation

1. Beschreibung des Produkts

Die Website «can-I-go.ch» entstand im Modul «»DGUI - Dynamic User Interfaces» der Klasse IW TZ ZH 2015 an der Hochschule für Technik und Wirtschaft HTW Chur als Gruppenprojekt unter Beteiligung aller Modulteilnehmenden. Als Tool zur kollaborativen Zusammenarbeit und Entwicklung wurde die Plattform GitHub in Kombination mit Travis CI verwendet. Ziel war, inkrementell eine Website zu entwickeln, welche Interessierten Empfehlungen zu geplanten Freizeitaktivitäten liefert, indem sie die dazu notwendigen Web-APIs abfragt. Beispielsweise sollte es einem User möglich sein, herauszufinden, ob heute das Windsurfen auf dem Walensee unter Berücksichtigung der aktuellen Wetterlage möglich ist. Die Website deckt die folgenden Aktivitäten ab: Schwimmen in einem stehenden Gewässer, Joggen in einem Wald, Windsurfen auf einem stehenden Gewässer, auf einen Hügel oder Berg wandern.

Die für das Projekt zentralen Dateien befinden sich im GitHub-Repository (https://github.com/hauserulrich/FS19_IW15tz_Zuerich_DGUI) im Ordner «www». Es sind dies: index.html im Unterordner «html», main.js im Unterordner «js», main.css im Unterordner «style». Die benötigten Bilder sind abgelegt im Order «images».

Zur Entwicklung verwendet wurden JavaScript (ECMA Script 6) , HTML 5, CSS 3 sowie die Frameworks jQuery 3.2.1 und Bootstrap 4.0. Die Website erfragt Daten von folgenden REST-APIs:

- HERE-API[1]: Endpoints für Karte[2] und Wetterdaten[3]
- Schweizer Instanz[4] für die Overpass-API (OpenStreetMap[5]): mögliche Orte für die gewählten Aktivitäten

Ursprünglich war von den Dozierenden vorgesehen, ausschliesslich die HERE-API zu benutzen. Für die Abfrage der Orte für die gewählten Aktivitäten hat sich im Laufe des Projekts aber die Overpass-API zu OpenStreetMap als praktikabler erwiesen. Die Website wurde hauptsächlich getestet mit dem Webbrowser Google Chrome.

Bei der Website handelt es sich um eine Single Page Application. Der Body besteht aus drei Teilen: Dem Input-Formular, der Ergebnisanzeige in Form der Karte (map) sowie der Wetter-Anzeige. Das Input-Formular (vgl. index.html, Zeile 37) ist in ein jumbotron-Element (vgl. index.html, Zeile 17) integriert und fragt den Uster nach den Angaben zur gewünschten Aktivität (Ort, Aktivität, Datum). Zusätzlich kann der User Filter (u.a. Umkreis) auswählen (vgl. index.html, Zeilen 121-140). Die zu den Eingaben des Users

[1] https://developer.here.com [09.07.2019]
[2] https://developer.here.com/documentation#maps [09.07.2019]
[3] https://developer.here.com/documentation#weather [09.07.2019]
[4] http://overpass.osm.ch [09.07.2019]
[5] https://www.osm.ch/ [09.07.2019]

passenden Treffer werden in der Karte (vgl. index.html, Zeilen 152-156) oder wahlweise in Tabellenform (vgl. index.html, Zeilen 160-162) angezeigt. Die Wetter-Anzeige (vgl. index.html, Zeilen 166-173) besteht aus sechs dynamisch generierten card-Elementen, die zu fix ausgewählten Schweizer Städten (Bern, Genf, Lugano, Chur, Luzern, St. Gallen) das aktuelle Wetter sowie ein statisch eingebundenes Foto anzeigen.

2. Funktionalitäten

Nach dem Aufbau der Seite wird *document.ready(function)* aufgerufen (Zeile 31). Die in Zeile 32 definierte Funktion *cityLoop()* (Zeilen 74-80) fragt für die sechs vordefinierten Städte (*cities*) unter Verwendung der Funktion *getWeatherJSONForCityLoop(chosenCity)* (Zeilen 185-213) über den Wetter-Endpoint der HERE-API die aktuellen Wetterdaten ab. Die Funktion *getWeatherJSONForCityLoop()* ruft nach Erhalt der API-Response (Zeile 205) die Funktion *createCard(chosenCity)* (Zeilen 907-975) auf, welche für die entsprechende Stadt (*chosenCity*) ein card-Element mit den aktuellen Wetterdaten sowie dem jeweiligen statischen Bild erzeugt. Innerhalb der *document.ready()*-Funktion wird in Zeile 38 dem submit-Button des Eingabeformulars (vgl. index.html, Zeile 114) ein EventLister *'submit'* hinzugefügt und in Zeile 39 das Datum im Eingabeformular (vgl. index.html, Zeile 56) per default auf den heutigen Tag gesetzt. In Zeile 51 werden die Ausgangskoordinaten für die Karte definiert, auf deren Basis die Funktion *getMap(currentCenterCoordinates)* (Zeilen 978-1002) anschliessend über die HERE-API die Karte abfragt und zentriert. Durch den zu *window* hinzugefügten EventListener *'resize'* (Zeile 61) wird sichergestellt, dass die Karte den gesamten dafür vorgesehenen Container beansprucht.

Gibt der User Daten in das Eingabeformular ein und betätigt den submit-Button, wird die function *handleSubmit(e)* aufgerufen (Zeilen 91-157). Diese prüft alle Formulareingaben bzw. -filter auf ihre Existenz und fragt sie, falls vorhanden, ab. Anschliessend (Zeilen 147-153) wird geprüft, ob das gewählte Datum auf den heutigen Tag fällt. Trifft dies zu, wird die Funktion *getWeatherObservationJSON(userCityInputString)* aufgerufen. Liegt das gewählte Datum in der Zukunft, resultiert der Aufruf der Funktion *getWeather7DayForecastJSON(userCityInputString)*.

Die Funktion *getWeatherObservationJSON()* (Zeile 216-264) macht eine AJAX-Abfrage des Wetter-Endpoints der HERE-API und holt sich die aktuellen Wetterdaten für den ausgewählten Ort. Da die HERE-API für einen Ort meist mehrere Treffer hat, wird in Zeile 231 der Treffer mit Distanz 0 ausgewählt. Die Koordinaten des ausgewählten Orts werden in der Variable *coordinatesForActivities* gespeichert. Basierend auf diesen Koordinaten wird die Karte mittels Aufruf der Funktion *map.setCenter(coordinatesForActivities)* neu ausgerichtet. In Zeile 249 wird die Variable *noForecast* auf *false* gesetzt, da für den betreffenden Ort Wetterdaten gefunden worden sind. Schliesslich werden Koordinaten, Umkreis und ausgewählte Aktivität an die nächste Funktion

getOverpassTransformedActivityList(coordinatesForActivities, rangeInput, userActivityInputString) übergeben. Die Funktion *getWeather7DayForecastJSON()* (Zeile 267-321) führt die gleichen Schritte wie die Funktion *getWeatherObservationJSON()* aus, fragt jedoch den HERE-Endpoint für den 7-Tage-Forecast ab. Liegt das gewählte Datum weiter als sieben Tage in der Zukunft bleibt die Variable *noForecast* auf *true* gesetzt (Zeile 308). Dies führt später (Zeilen 691-700) zu einer entsprechenden Information an den User.

Die Funktion *getOverpassTransformedActivityList (coordinates, rangeInput, activityInputString)* (Zeilen 327-452) erstellt mit Hilfe von *switch case* eine Liste (*transformedActivityList*) mit den Kategorien, welche für eine bestimmte Aktivität (z.B. «swimming») über die Overpass-API abgefragt werden sollen. Dies ist notwendig, weil die Overpass-API für eine vom User gewählte Aktivität Daten mit unterschiedlichen Kategorien (*categoryName*) bereithält. So sind bei der Aktivität «swimming» sowohl Daten mit *categoryName 'lake', 'pond'* als auch *'stream_pool'* einzubeziehen (Zeilen 338-348). Bei der Aktivität «surfing» wurde, um die ansonsten sehr kleine Treffermenge etwas zu vergrössern, auch die Kategorien mit *categoryName 'pond'* und '*stream_pool'* miteinbezogen. Die erstellte Liste der Kategorien *(transformedActivityList)* wird in der Variable *activityList* gespeichert (Zeile 443). Anschliessend wird die nächste Funktion *getOverpassInterestingNodesAround(chosenCityCoordinates, rangeInput)* aufgerufen.

Die Funktion *getOverpassInterestingNodesAround(coordinates,range)* (ab Zeile 457) findet mögliche Orte für die Aktivitäten im Umkreis (*range*) des gewünschten Orts (*coordinates*). In den Zeilen 470-496 wird die für die anschliessende Abfrage der Overpass-API notwendige URL (*queryString*) zusammengesetzt. Der Vollständigkeit halber werden *nodes, ways* und *relations* abgefragt. Die URL für die Aktivitäten «swimming» und «surfing» mit *categoryID==='water'* muss separat zusammengesetzt werden, weil die URL zusätzlich das tag *"natural"="water"* für die natürlichen Gewässer enthalten muss. Schliesslich wird in der AJAX-Abfrage (ab Zeile 502) die Overpass-API mit der erstellten URL abgefragt. Jedes gefundene Element (*data.elements*) der Response wird dem Array *interestingPlaceList* hinzugefügt (Zeilen 512-518). In den Zeilen 520-550 werden unter Verwendung der Funktion *getDistance()* (Zeilen 161-182) die Distanzen zwischen dem Ursprungsort und den gefundenen Orten berechnet und für die Darstellung in der Tabellenansicht aufsteigend sortiert. Anschliessend wird die Funktion *renderlist(interestingPlaceList)* (Zeilen 1137-1160) aufgerufen, die nach Leeren der tabellarischen Ergebnisliste (div-Element mit id = *'resultlist'*) dem entsprechenden div-Element einen Button hinzufügt, der nach dem Klicken die Ergebnisse der aktuellen Anfrage tabellarisch darstellt (Zeilen 1145-1158).

In Zeile 557 wird die Funktion *checkConditionsForActivity(interestingPlaceList)* (ab Zeile 579) aufgerufen. Diese überprüft mittels Aufruf (Zeile 585) der Funktion *checkIfFiltersNotViolated()* (ab Zeile 707) zunächst, ob die im Eingabeformular definierten

Filterbedingungen eingehalten werden. Falls die Filter verletzt werden, weist die Funktion *checkIfFiltersNotViolated()*den Variablen *recommendation*, *cardTitle* und *cardDescription* die entsprechenden Werte zu und gibt die Information über die Funktion *get_recommendation_card(recommendation, cardTitle, cardDescription)* (Zeilen 1109-1135) aus. Falls die Filter eingehalten werden, wird ab Zeile 587 u.a. mit Hilfe von *switch case* geprüft, ob die für die gewünschte Aktivität notwendigen Konditionen (Temperatur (*forecastForSelectedDate.comfort*) für «swimming», «jogging», «hiking» bzw. Windgeschwindigkeit (*forecastForSelectedDate.beaufortScale)* für «surfing») erfüllt sind. Den Variablen *recommendation*, *cardTitle und cardDescription* werden entsprechende Werte zugewiesen. Über den Aufruf (Zeile 686) der Funktion *get_recommendation_card(recommendation, cardTitle, cardDescription)* (Zeilen 1109-1135) wird die Information als *'card'*-Element mit Bild angezeigt. Sind für das gewählte Datum keine Wetterdaten zur Verfügung (*noForecast === true*), werden lediglich die möglichen Orte für die Aktivität angezeigt sowie ein entsprechender Hinweis in der Karte (Zeilen 691-700). Die Darstellung der Orte für die Aktivität erfolgt in allen Fällen über den Aufruf (Zeilen 687 und 693) der Funktion *addInfoBubbleForOSM(interestingPlaceList)*. Diese Funktion (ab Zeile 875) erstellt eine neue Gruppe von Markern (*markerGroup*), die mit einem Event-Listener (*'tap'*) versehen sind, so dass bei jedem Klick auf einen Marker textuelle Zusatzinformation in der Karte aufpoppt. Anschliessend erfolgt in Zeile 899 der Aufruf der Funktion *generateMarkers(placesList).* Diese (Zeilen 1008-1060) generiert die beim Klick auf den Marker anzuzeigende Information (*htmlforMarker*) für jedes Element der Liste (*placesList*), d.h. für jeden Ort bzw. jeden Marker. Ebenso werden die jeweiligen Koordinaten der Elemente abgefragt (*coordinatesForMarker*). In Zeile 1057 wird schliesslich die Funktion *addMarkerToGroup(markerGroup, coordinatesForMarker, htmlForMarker)* aufgerufen. Die Funktion *addMarkerToGroup()* (Zeilen 1069-1103) setzt für jeden gefundenen Ort an den zugehörigen Koordinaten auf der Karte den Marker inkl. des beim Event *'tap'* anzuzeigenden Texts. Ausserdem versieht sie die entsprechenden Markern mit den passenden Icons, die als .png-Datei im Projekt vorliegen.