Weather App "weatherFLOSScast"

WP-Seminar "Quelloffene Software in der modernen Informatik" Projekt

Unter Beaufsichtigung von Dr. Chris Zimmermann

Frankfurt University of Applied Sciences

Brychcy, Patryk
patryk.brychcy@stud.fra-uas.de

Filatoff, Michael
michael.filatoff@stud.fra-uas.de

Fluegel, Dwipa
dwipa.fluegel@stud.fra-uas.de

Gavrilov, Sascha
sascha.gavrilov@stud.fra-uas.de

Karaman, Deniz
deniz.karaman@stud.fra-uas.de

Lepore, Dominik
dominik.lepore@stud.fra-uas.de

Zimmermann, Alwin
alwin.zimmermann@stud.fra-uas.de

Zundel, Benedikt
benedikt.zundel@stud.fra-uas.de

Friday, 2025-06-20

https://github.com/bzundel/weather-flosscast

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Technische Dokumentation

1.1 Datenvorbereitung (Benedikt Zundel)

Als Schnittstelle für die Wetterdaten haben wir uns für die `open-meteo` [1] API entschieden, da diese alle Daten liefert die wir anzeigen wollten und über weitere Funktionalitäten wie geocoding verfügt. Um die Daten zu konsumieren, definierten wir zunächst ein Datenstruktur (vgl. listing 1) um die abgeholten Daten intern zu repräsentieren. Die Idee war es, dem frontend für den Konsum nur ein einzelnes Objekt zu übergeben, über das sinnvoll iteriert werden kann.

Listing 1: Forecast Datenstruktur

```
data class Forecast(
    val timestamp: LocalDateTime,
    val days: List<DailyForecast>,
    val units: Units
)

data class DailyForecast(
    val date: LocalDate,
    val hourlyValues: List<Hourly>,
    val sunrise: LocalDateTime,
    val sunset: LocalDateTime
)

data class Hourly(
    val dateTime: LocalDateTime,
    val temperature: Double,
    val relativeHumidity: Int,
    val precipitationProbability: Int,
    val rain: Double,
    val showers: Double,
    val snowfall: Double,
    val weatherCode: Int
)

data class Units(
    val temperature: String,
    val humidity: String,
    val precipitationProbability: String,
    val rain: String,
    val showers: String,
    val snow: String,
)
```

Mittels Herangehensweisen aus der funktionalen Programmierung wie `map` lässt sich das von der API gelieferte JSON-Objekt mit wenig Aufwand, lesbar und modifizierbar in unsere Datenstruktur parsen (vgl. listing 2).

Listing 2: Auszug des Parsingprozesses

```
val hourly: JsonObject = json["hourly"]!!.jsonObject
val hourlyTime =
    hourly.getOrThrow("time").jsonArray.map { LocalDateTime.parse(it.jsonPrimitive.content)
    }
val hourlyTemperature =
    hourly.getOrThrow("temperature_2m").jsonArray.map { it.jsonPrimitive.content.toDouble()
    }
...
val dailyForecasts: List<DailyForecast> = hourlyValues.groupBy { it.dateTime.date }
    .map { (date, measurements) -> DailyForecast(date
        , measurements
        , sunrise.first { it.date == date
    }.toInstant(TimeZone.UTC).toLocalDateTime(currentTimezone)
        , sunset.first { it.date == date
    }.toInstant(TimeZone.UTC).toLocalDateTime(currentTimezone)) }
```

Die Abfrageadresse ergibt sich aus Koordinaten die von Nutzer*innen als ausgewählte Stadt oder aktuellem Standort übergeben werden. Zur Realisierung der Städtewahl wird die selbe API unter dem geocoding

Endpunkt angesprochen. Dieser liefert auf Basis eines (eventuell unvollständigem) Städtenamen eine Liste an bekannten Städten als Vorschläge, die dann in einem ähnlichen Prozess von JSON in eine von uns definierte Datenstruktur geparsed werden und dem frontend zum Anzeigen weitergegeben wird.

In dem gesamten Abfrageprozess wird hoher Wert auf die Richtigkeit der Daten und eine erfolgreiche Abwicklung der Anfragen gelegt. Viele der Schritte werden während der Verarbeitung der Daten gegen Erwartungswerte geprüft und unerwartete Situationen werden mittels Exceptions unmittelbar an das frontend kommuniziert. In Fehlersituation, sowie beim normalen Ablauf, wird eine Nachvollziehbarkeit des Prozesses mittels intensiven Loggings erreicht.

1.2 Caching **(Benedikt Zundel)**

Unnötige Abfragen werden zum Schutze des Verbrauchs der Nutzer*innen, sowie zur Entlastung der API unterbunden. Ein weiterer Nutzen ist die Verfügbarkeit der aktuellen Daten ohne Internetverbindung. Um das zu realisieren, implementieren wir ein dateibasiertes Caching-System. Die unterliegende Datei enthält die von der API abgeholten Wetterdaten nach unserer definierten Datenstruktur serialisiert in einem JSON-Objekt, welches die Daten via gekürztem Koordinatensatz indiziert. Durch das Kürzen der Koordinaten auf eine Nachkommastelle erreichen wir einen klein gehaltenen Datensatz der gecached wird, da nicht jede minimale Veränderung im Standort als neuer individuelle Standort angesehen wird. Der Standort wird durch die Kürzung auf circa $12km \times 12km$ Kacheln abgebildet, was, durch die kleine Fläche, zu keinem Genauigkeitsverlust führt. Bei einem Aufruf der vom backend offengelegten Schnittstelle wird zunächst der Cache geladen und geprüft ob für die angefragten Koordinaten bereits ein Eintrag existiert. Sollte dies nicht der Fall sein, wird eine Abfrage auf die API gemacht und die Daten werden im Cache gespeichert. Wenn ein Eintrag gefunden wird, wird das Alter der Daten über ein Zeitstempel Feld geprüft. Ist ein Eintrag älter als eine Stunde, gilt dieser als veraltet und wird aktualisiert. Sonst wird der Eintrag aus dem Cache geladen und es geschieht keine Abfrage an die API.

Literatur

[1] Open-meteo: Free weather api. https://open-meteo.com/. (Accessed: 2025-06-14).