Lehrstuhl für Pervasive Computing Systems

Teco

Tobias Röddiger
Dr. Paul Tremper

Anwenderorientierte Nutzerschnittstelle für Luftqualitätsdaten

Implementierung

Anna Csurkó
Jona Enzinger
Yannik Schmid
Jonas Zoll

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument dokumentiert die Änderungen an dem Softwareprojekt
Anwenderorientierte Nutzerschnittstelle für Luftqualitätsdaten
während der Implementierung. Diese ist über einen Zeitraum von 4 Wochen entstanden wobei der Implementierungsplan nach Wochen gegliedert war.

Im Kapitel *Änderungen* wird genauer auf die Entwurfsentscheidungen eingegangen die aus verschiedenen Gründen nicht in der Software umgesetzt wurden.
Im Kapitel *Herausforderungen* werden Verzögerungen gegenüber der Planung und ihre Ursachen erwähnt, beispielsweise Eigenheiten von React und TypeScript die beim Entwurf nicht bekannt waren.

Es wurden alle Muss- und Wunschkriterien implementiert.

2 Herausforderungen

2.1 MapConfigurationMemory

Diese Klasse erfüllt die Aufgabe die aktuell gewählte Konfiguration und Viewport in den Local Storage zu schreiben und später wieder zu laden. Bei der testweisen Implementierung während der Entwurfsphase gab es hierbei keine Probleme. Da dort aber lediglich mit Interfaces als Speicherdaten gearbeitet wurde und nun Klassen vorhanden sind ergaben sich neue Schwierigkeiten. So können die Objekte nun nicht mehr nur mit der JSON Klasse de-/serialisiert werden sondern müssen vom Programm erstellt werden und mit einzelnen Werten aus dem `Local Storage` befüllt werden. Da dies leichte Anpassungen im `MapController` erforderte verzögerte sich hier der Zeitplan geringfügig.

2.2 FeatureProvider

Hier führte eine Eigenheit von React / Webpack zu erheblichen Verzögerungen: require('...') kann nur mit festen Zeichenketten problemlos funktionieren sofern die benötigten Dateien nicht im selben Ordner liegen. Nach einiger Recherche werden die Definitionen nun mit require.context(...) geladen, was allerdings einen großen Nachteil mit sich bringt. Jest unterstützt diesen Befehl nicht was dazu führt dass alle Tests abstürzen die irgendein Feature benutzen wollen. Da das gerade mit Hinblick auf die Qualitätssicherungs-Phase nicht tragbar ist wurden letztendlich alle Features in einer feature.json als Liste gespeichert und beim Anwendungsstart geladen. So wird nur wenig Flexibilität eingebüßt.

3 Änderungen

3.1 FeatureProvider

FeatureProvider

Die Funktionalität dieser Klasse sollte ursprünglich in Controller.Frost.DataProvider enthalten sein und nach außen versteckt. Dies ermöglicht außerdem den Zugriff auf die Features auch außerhalb des FROST-Pakets.

Methoden

- `static getInstance()`
  Liefert die Singleton-Instanz zurück oder erstellt sie.
- `constructor()`
  Initialisiert den Feature-Speicher und lädt die Definitionen
- `getFeature(id : string) : Feature|undefined`
  Das gespeicherte Feature falls es existiert. Schlägt dies fehl wird 'undefined' zurückgegeben.

3.2 MapController

MapController

Skala und Viewport werden von außen lesbar gemacht. Damit muss MapPage keine eigene Kopie bereithalten.

Hinzugefügt

- `getScale() : Scale`
  Die aktuelle Skala.
- `getViewport(): Viewport`
  Der aktuelle Viewport.

3.3 OnSearch(string) für Suche

OnSearch(string) für Suche

Die eigentliche Suche findet nun außerhalb der Komponente statt. Damit wird die Komponente leichter für verschiedene Anwendungen wiederverwendbar.

Hinzugefügt

- `Search.Props.onSearch(term: string) : void`
  Wird bei Klick auf den Such-Button oder Drücken von Enter aufgerufen. Enthält den aktuellen Inhalt der Suchbox.
- `MapView.onSearch(term: string) : void`
  Ruft die Suche im MapController auf und aktualisiert die Seite.

3.4 Legend

Legend

Der Ausschnitt der von der Legende angezeigt wird soll flexibel sein.

Hinzugefügt

- `Props.min : number`
  Das untere Ende der Legende
- `Props.max : number`
  Das obere Ende der Legende

3.5 Map

Map

Ursprünglich sollte der Viewport über die Mitte der Pins/Polygone bestimmt werden. Um die Karte von Anfang an auf die letzte Position zu zentrieren wird der Viewport direkt übergeben.

Hinzugefügt

- `Props.viewport : Viewport`
  Viewport mit dem die Karte initialisiert wird.

3.6 FeatureSelectInit

FeatureSelectInit

Die FeatureSelect Auswahl benötigt die aktuellen Werte um sie standardmäßig auswählen zu können.

Hinzugefügt

- `FeatureSelect.Props.startConf?: { conf: string; feature: string }`
  Die Startwerte der Auswahlboxen. Optional.
- `MapController.getFeatureSelectConf(): conf: string; feature: string`
  Gibt Werte für die Auswahlboxen basierend auf der Konfiguration aus.

3.7 Diagramme der Detailansicht

Diagramme der Detailansicht

Im ursprünglichen Entwurf gab es eine abstrakte Diagrammklasse, die in unserer MVC-Architektur in der View zu verorten war. Unterschiedliche Diagramme sollten als unterschiedliche Klassen, die alle von der abstrakten Diagrammklasse erben, implementiert werden. Mit diesem Entwurf sind wir nun auf Probleme gestoßen, da React empfiehlt keine Vererbung von Komponenten zu verwenden, wodurch wir gezwungen waren unsere Modellierung zu verändern.
Im Zuge dessen haben wir uns entschieden die Vererbung durch Komposition zu ersetzen. Nun gibt es weiterhin eine Diagrammklasse, die allerdings nun nicht abstrakt, sondern eine React Komponente ist, deren Aufgabe alleine im Anzeigen eines Diagramms besteht. Da wir zur Darstellung von Diagrammen React Google Charts"verwenden dient die Diagrammklasse also nur als Adapter der Chart Komponente dieser Bibliothek.
Die Differenzierung unterscheidlicher Diagrammtypen wird durch Komposition unterschiedlicher DiagramController erzielt, die einer Diagramm Komponente in den props übergeben werden. Ein DiagramController implementiert eine entsprechende Schnittstelle. Seine Aufgaben bestehen im Anfordern der diagrammspezifischen Daten vom DataProvider, dem Formatieren der Daten in ein zur Darstellung geeignetes Format und dem Wechsel zwischen unterschiedlicher Konfigurationsoptionen.
Des Weiteren haben wir eine DiagramFactory Klasse mit einer statische Fabrikmethode ergänzt. Über diese ist es möglich mit einer typenId einen DiagrammController für ein bestimmtes Feature einer Messstation zu erzeugen. Diese Klasse hat im ursprünglichen Entwurf gefehlt, wodurch die Aufgabe im Model hätte erfolgen müssen, was allerding der MVC-Architektur widerspricht.
Insgesamt harmoniert dieser neue Entwurf sehr viel besser mit React und unserer MVC-Architektur und vereinfacht zusätzlich den Code innerhalb der Diagrammklasse. Des Weiteren ist die Kopplung zwischen Inhalt und Darstellung eines Diagramms im neuen Entwurf sehr viel loser, wudurch es einfacher ist die Applikation durch neue Diagrammtypen zu ergänzen oder bestehende Diagrammtypen abzuändern.
Das Klassendiagramm des neuen Entwurfs sieht nun wie folgt aus:

3.8 IDs für MapConfiguration

IDs für MapConfiguration

Ursprünglich sollte die Art der Konfiguration über das name Attribut bestimmt werden. Das führt aber im build zu Problemen da Klassennamen komprimiert werden.
Hinzufügen

- `abstract getId() : string`
  Die ID der Karten-Konfiguration

3.9 Featurespezifische Icons

Featurespezifische Icons

Jedes Feature besitzt nun eine eigenes Icon. Das Icon wird über den Konstruktor übergeben.
Hinzufügen

- `abstract getId() : string`
  Die ID der Karten-Konfiguration

3.10 ObservationItem

ObservationItem

Um den Code in der Klasse ObservationStationProfile zu vereinfachen haben wir eine Klasse ObservationItem ergänzt. Ein ObservationItem ist ein Punkt in der Liste der letzten gemessenen Werte. Als reine View Komponente zeigt ein ObservationItem also einen Messwert, das jeweilige Feature und ein featurespezifisches Icon an.

3.11 App Configuration

App Configuration

Es wurde eine eigene Klasse für die Anwendungskonfiguration erstellt (Singleton) um eventuelle neue Einstellungen möglichst einfach hinzufügen zu können. In einer config.json befindet sich ein Objekt des Typs IConfig **Hinzugefügt**

- `Configuration.getInstance()`
  Lädt die Konfigurationsdatei in das Singleton-Objekt
- `Configuration.getLanguage()`
  Die Standard
- `Configuration.getFrostUrl()`
  Das Top-Level der FROST-REST-API
- `interface IConfig`
  frostUrl: string;
  language: string;
  supportedFeatures: string[];

3.12 IDs für MapConfiguration

IDs für MapConfiguration

Um die verschiedenen Konfigurationen ohne fehleranfällige typeof Abfragen unterscheiden zu können wurden IDs hinzugefügt die i.d.R. als Konstante gespeichert sind.

- **MapConfiguration.getID() : string**
  Eine eindeutige Kennung der MapConfiguration nach Typ.