



FH Aachen

**Fachbereich
Elektrotechnik und Informationstechnik
Studiengang Informatik**

Bachelorarbeit

Konzeption und prototypische Entwicklung einer Webanwendung für
Messdatenerhebung und -bereitstellung

Tuan Anh Cong Nguyen
Matr.-Nr.: 3517392

Referent: Prof. Dr. rer. nat. Heinrich Faßbender

In Zusammenarbeit mit Ausbildungsbetrieb:
ahu GmbH Wasser Boden Geomatik

Externer Betreuer: Dr. David Loibl

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, den 23. Mai 2025

.....

Inhaltsverzeichnis

Abstract	6
1 Einleitung	7
1.1 Motivation der Arbeit	7
1.2 Ziel der Arbeit	7
1.3 Aufbau der Arbeit	8
2 Technologischer Überblick	9
2.1 Vaadin	9
2.2 Spring Framework	10
2.3 Spring Boot	10
2.4 Maven	11
2.5 Vaadin Initializer	12
2.6 Liquibase	12
2.7 Datenbankmanagementsystem	15
3 Anforderungsanalyse und Spezifikation	16
3.1 Analyse	16
3.1.1 Anwendungsszenario	16
3.1.2 Stakeholder und User-Stories	16
3.2 Spezifikation	17
3.2.1 Use-Cases	17
4 Konzeption und Architektur	22
4.1 Konzeption	22
4.2 Design-Entscheidung	27
5 Implementierung	28
5.1 Übersicht über wichtige Klassen und Entitäten	28
5.2 Front-End-Komponente	33
6 Fazit und Ausblick	48
6.1 Bewertung	48
6.1.1 Bewertung von Implementierung	48

Inhaltsverzeichnis

6.1.2 Bewertung von Architekturdesign und Technologien	49
6.2 Ausblick	50

Abkürzungsverzeichnis

REST-API	Representational State Transfer Application Program Interface
OGC	Open Geospatial Consortium
WMS	Web Map Server
WFS	Web Feature Server
JSON	JavaScript Object Notation
CSV	Comma Separated Values
ASCII	American Standard Code for Information Interchange
QGIS	Quantum Geographic Information System
UI	User Interface
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
AJAX	Asynchronous JavaScript and XML
J2EE	Java 2 Platform, Enterprise Edition
XML	Extensible Markup Language
JAR	Java Archive
WAR	Web Application Archive
JPA	Java Persistence API
CRUD	Create, Raed, Update and Delete
UUID	Universally Unique Identifier

Abstract

In einer zunehmend digitalisierten und automatisierten Welt wächst die Menge neu generierter Geodaten – insbesondere Mess- und raumbezogene Daten - kontinuierlich.

Die Recherche nach relevanten Messdaten wie Grundwasserständen, Pegelständen und Klimadaten (Niederschlag, Temperatur), sowie deren Umformatierung in ein einheitliches Format stellt eine erhebliche Herausforderung dar. Dies liegt vor allem an der Vielzahl unterschiedlicher Datenanbietern, die jeweils eigene Datenstrukturen und -formate verwenden.

Die Bereitstellung dieser Rohdaten erfolgt häufig im CSV-Format oder als individuell strukturierte ASCII-Dateien. Im Kontext der Ahu GmbH werden diese Daten über den sogenannten **ahuManager** konsumiert. Vor der Weiterverarbeitung müssen die Rohdaten jedoch in ein kompatibles Zielformat umgewandelt werden – ein Vorgang, der bislang teilweise manuell (z.B. mithilfe von Excel) durchgeführt wird. Diese manuelle Datenaufbereitung ist zeitaufwendig, fehleranfällig und nur begrenzt skalierbar.

Ziel dieser Arbeit ist es daher, zu untersuchen, wie eine webbasierte Applikation zur automatisierten Umwandlung und Bereitstellung von Messdaten konzipiert und prototypisch umgesetzt werden kann.

1 Einleitung

1.1 Motivation der Arbeit

Diese Bachelorarbeit entsteht in Kooperation mit dem Bereich Geomatik der Ahu GmbH, der sich auf die Konzeption und Entwicklung von Monitoringsysteme sowie Webanwendungen spezialisiert hat. Ziel ist es, Dienstleitern, Betreibern und Aufsichtsbehörden ein kosteneffizientes Management von Geodaten - wie etwa Grundwasser-, Oberflächenwasser-, und Bodendaten- zu ermöglichen.

Aktuell besteht ein Bedarf an einem zentralen System, das eine automatisierte Suche, Aggregation und Verwaltung von Geodaten aus unterschiedlichen externen Datenquellen ermöglicht. Ziel ist es, diese heterogenen Daten in eine homogene Struktur zu überführen und sie einem bestehenden Client-System, wie dem **AhuManager**, standardisiert bereitzustellen.

1.2 Ziel der Arbeit

Das Ziel dieser Bachelorarbeit besteht in der Konzeption und prototypischen Umsetzung einer webbasierten Anwendung zur Erhebung und Bereitstellung von Messdaten. Ein zentrales Anliegen ist die Entwicklung einer übergreifenden Suchfunktion für Geodaten sowie deren Umwandlung in ein definiertes Zielformat.

Durch die Umsetzungen dieser Funktionalitäten soll perspektivisch eine Software entstehen, die eine praktikable Lösung für bestehende Herausforderungen in der Verarbeitung und Bereitstellung heterogener Messdaten bietet.

1.3 Aufbau der Arbeit

Kapitel 1: Einleitung

Im ersten Kapitel werden die Motivation sowie die Zielsetzung der Arbeit erläutert. Es wird auf die Relevanz des gewählten Themas eingegangen, welchen potenziellen Nutzen die prototypische Entwicklung für das Unternehmen bieten kann.

Kapitel 2: Technologischer Überblick

Dieses Kapitel stellt die in der Entwicklungsphase eingesetzten Technologien und relevanten Begriffe vor - von Bibliotheken zur Verfolgung der Datenbankänderungen über das verwendete Backend-Framework bis hin zum verwendeten UI-Framework für die Benutzeroberfläche.

Kapitel 3: Anforderungsanalyse und Spezifikation

In diesem Kapitel werden verschiedene User-Stories definiert und daraus Use-Cases abgeleitet. Die funktionalen sowie nicht-funktionalen Anforderungen an das System werden spezifiziert und dokumentiert.

Kapitel 4: Konzeption und Architektur

In diesem Kapitel wird das zugrunde liegende Architekturdesign des Prototyps aus unterschiedlichen Sichten erläutert. Die getroffenen Architekturentscheidungen werden begründet und mit gängigen Architekturmustern verglichen.

Kapitel 5: Implementierung

Die tatsächliche Umsetzung des Prototyps wird anhand ausgewählter Codebeispiel und technischer Erläuterung detailliert dargestellt. Dabei wird auf zentrale Aspekte wie Datenverarbeitung, Schnittstellenlogik und UI-Integration eingegangen.

Kapitel 6: Fazit und Ausblick

Abschließend erfolgt eine kritische Bewertung der erreichten Ergebnisse im Hinblick auf die zu Beginn definierten Anforderungen. Zudem wird ein Ausblick auf möglich Weiterentwicklung und Optimierungspotenziale gegeben.

2 Technologischer Überblick

2.1 Vaadin

Vaadin ist ein serverseitiges Web-Framework, das dem Entwickler erlaubt, moderne Webanwendung in Java zu entwickeln, ohne dass explizit HTML,CSS oder JavaScript geschrieben werden muss. Vaddin verfügt über eine große Komponentenbibliothek, die eine Vielzahl an vorgefertigten UI-Elementen wie Buttons, Tabellen, Formulare, Dialoge und Layouts bietet.

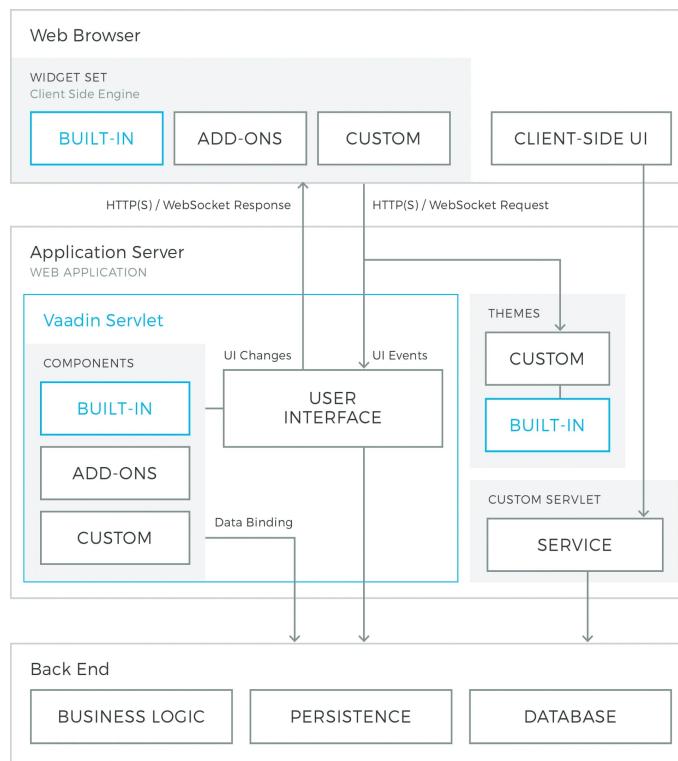


Abbildung 2.1: Übersicht Vaadin Architektur Document, [2021](#)

Vaadin verfolgt einen serverseitigen Rendering-Ansatz. Während eine AJAX-basierte Vaadin Client-Side Engine dafür sorgt, dass die Benutzeroberfläche im Browser durch ein Widget-Set gerendert wird. Die Benutzeroberfläche kann aus eingebauten Komponenten, Add-Ons und benutzerdefinierten Komponenten bestehen. Diese Client-Side Engine kommuniziert über HTTP oder WebSockets mit dem Server. Die gesamte UI-Logik wird dann auf dem Server ausgeführt. Die Vaadin Servlet empfängt Client-Anfragen und aktualisiert die Benutzeroberfläche. Die UI-Komponenten werden serverseitig erstellt und verwaltet. Änderungen

in der UI werden durch die Vaddin Client-Side Engine an den Browser zurückgespielt.

2.2 Spring Framework

Spring Framework ist ein Java-Framework, das um 2003 als Reaktion auf die damals noch zu komplizierte J2EE-Plattform entwickelt wurde. Spring ermöglicht eine einfachere und unkompliziertere Entwicklung von Enterprise-Applikation in Java. Eines der wichtigsten Konzepte von Spring war die Inversion of Control (IoC). IoC ist ein Prinzip, bei dem der Kontrollfluss an eine externe Quelle(z.B. ein Framework) übergeben wird. Das Framework ist dann gemäß einer Spezifikation für die Erstellung und Löschung der Objekte und den Aufruf von Methoden verantwortlich.

Spring führt das Konzept des Bean-Containers ein. Beans sind Java-Objekte, die von Spring instanziert und verwaltet sind. Wenn die Haupt-Bean von der Hilfs-Bean abhängig ist, stellt Spring sicher, dass die Hilfs-Bean vor der Haupt-Bean initialisiert wird. Spring injizierte außerdem die Instanz der Hilfs-Bean in die Haupt-Bean, sodass die Haupt-Bean nicht mehr nach ihren eigenen Abhängigkeiten suchen muss. Dieses Muster wird als Dependency Injection bezeichnet.

Damals musste der Entwickler die Bean-Konfiguration in XML schreiben, die Spring anweist, wie die Beans zu konstruieren sind. Inzwischen wird diese mithilfe einer Kombination aus Java-Annotation, Java-Code und Konventionen erstellt.

2.3 Spring Boot

Mit dem Wachstum der Spring-Plattform nahm auch die Komplexität der Entwicklung von Spring-Anwendungen zu. Gleichzeitig verbreitet sich der Einsatz von Microservices und containerisierten Umgebungen in der Softwareentwicklung. Entwickler wünschten sich einfach Methode, um schlanke Webanwendungen zu erstellen, die unabhängig als eigenständige Dienst ausgeführt werden können, anstatt auf einem dedizierten Anwendungsserver zu laufen.

Spring Boot wurde als Reaktion auf diese Anforderungen entwickelt. Es erleichtert die Erstellung von Spring-Anwendungen durch sinnvolle Standard-Einstellungen, Starter-Abhängigkeiten und produktionsreife Funktionen für Konfiguration und Überwachung. Durch die Einführung eingebetteter Servlet-Container wurde es möglich, Anwendungen als eigenständige, ausführbare JAR-Dateien zu verpacken.

2.4 Maven

Maven ist ein Build-Management- und Projektverwaltungswerkzeug, das häufig in der Java-Entwicklung eingesetzt wird. Es automatisiert Aufgaben wie das Herunterladen von Abhängigkeiten, das Kompilieren von Quellcode und das Ausführen von Builds sowie Tests. Maven verwendet eine zentrale Konfigurationsdatei, die `pom.xml`, um alle Aspekte des Projekts wie Libraries, Plugins und Build-Prozesse zu steuern.

Die Funktionsweise von Maven basiert auf einem deklarativen Ansatz. Das bedeutet, dass der Entwickler alle relevanten Informationen über das Projekt in einer zentralen Konfigurationsdatei, der sogenannten `pom.xml` (Project Object Model) angibt, wie z.B. Abhängigkeiten, Build-Prozesse und Plugins : Deinhard, 2024.

1. Die `pom.xml`-Datei (Project Object Model)

Die `pom.xml` ist das Herzstück eines Maven-Projekts. Sie enthält Informationen wie:

- Projektinformationen: Name des Projekts, Version, Beschreibung.
- Abhängigkeiten: Welche Bibliotheken und Frameworks das Projekt benötigt.
- Plugins: Zusätzliche Werkzeuge, die den Build-Prozess erweitern (z.B. Compiler-Plugins, Test-Frameworks).
- Repositories: Woher Maven externe Abhängigkeiten herunterladen soll, typischerweise das Maven Central Repository.
- Build-Spezifikationen: Kompilierungsanweisungen, Testkonfigurationen und Deployment-Optionen.

2. Build-Lifecycle

Maven besitzt einen vordefinierten Build-Lifecycle, der aus mehreren Phasen besteht. Zu den wichtigsten Phasen gehören:

- validate: Überprüft, ob alle erforderlichen Informationen im Projekt vorhanden sind.
- compile: Kompiliert den Quellcode.
- Führt automatisierte Tests aus.
- package: Verpackt den kompilierten Code in ein Distributionsformat, typischerweise eine JAR- oder WAR-Datei.

- install: Installiert das Paket in das lokale Maven-Repository, damit es in anderen Projekten verwendet werden kann.

2.5 Vaadin Initializer

Gegenüber dem üblichen [Spring Initializer](#), der ein vorkonfiguriertes Spring-Projekt bereitstellt, bietet der [Vaadin Initializer](#) den Vorteil, dass er speziell auf Vaadin-Projekte zugeschnitten ist und eine vereinfachte und schnellere Projektgenerierung ermöglicht, insbesondere wenn man Spring Boot als Backend verwendet.

Mit dem Vaadin- Initializer kann man direkt ein Vaadin Flow-basiertes Projekt mit einem Spring Boot-Backend erstellen, während man mit dem Spring Initializer ein allgemeines Spring Boot-Projekt generiert und dann manuell die Vaadin-Abhängigkeiten hinzufügen muss.

2.6 Liquibase

Dank Versionsverwaltung wie Git kann man in den meisten Projekten den Weg einer Code-Änderung von der Entwicklung über Test bis hin zum produktiven Deployment richtig verfolgen und nachvollziehen. Was für den Code gilt, sollte auch für die Datenbankanpassungen gelten. Liquibase ist eine Bibliothek um Änderungen an einem Datenbankschema verfolgen, verwalten und anwenden zu können. Mit folgendem Eintrag in pom.xml kann man in ein Spring-Projekt integrieren.

```
1 <dependency>
2   <groupId>org.liquibase</groupId>
3   <artifactId>liquibase-core</artifactId>
4   <version>4.23.1</version>
5 </dependency>
```

Für die Speicherung der Daten wird die PostgreSQL-Datenbank entschieden. PostgreSQL selbst eine leistungsstarke, offene Datenbank, die kompatibel zu Liquibase passt. Die Aktivierung der Datenbank wird in der Datei application.properties innerhalb der Spring-Applikation wie folgt konfiguriert:

```
1 spring.datasource.url=jdbc:postgresql://192.168.252.140:5432/opendataconn_ng
2 spring.datasource.driver-class-name=org.postgresql.Driver
3 spring.datasource.username=opendataconn_ng_usr
4 spring.datasource.password=
5 spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
6 spring.liquibase.change-log=classpath:/db/changelog-root.xml
```

Für die Verwendung von Liquibase sind folgende Dateien notwendig:

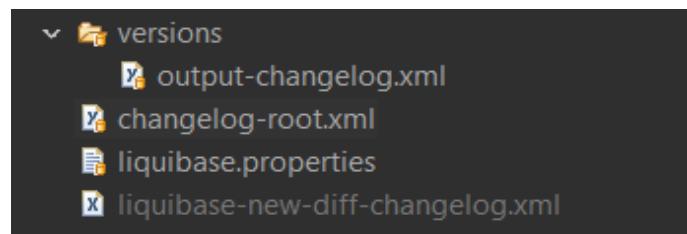


Abbildung 2.2: Liquibase-Konfiguration

1. Liquibase Konfiguration - liquibase.properties

In dieser Datei werden allgemeine Einstellungen wie der Datenbankzugang und die zu verwendenden Changelog-Konfiguration festgelegt.

```
1 # Database Configuration
2 url=jdbc:postgresql://192.168.252.140:5432/opendataconn_ng
3 username=opendataconn_ng_usr
4 password=
5 driver=org.postgresql.Driver
6 # Reference Configuration
7 referenceUrl=hibernate:spring:de.ahu.opendata?dialect=
8 org.hibernate.dialect.PostgreSQLDialect
9 referenceDriver=liquibase.ext.hibernate.database.connection.HibernateDriver
10 # Output Files
11 changeLogFile=src/main/resources/db/changelog-root.xml
12 diffChangeLogFile=src/main/resources/db/liquibase-new-diff-changelog.xml
13 outputChangeLogFile=src/main/resources/db/versions/output-changelog.xml
```

2. Changelog Konfiguration - changelog-root.xml

Ein so genannter Root-Changelog wird erstellt, welcher in XML geschrieben ist und eine output-changelog.xml inkludiert, welche im Ordner „versions“ zu finden ist. Diese Datei sorgt dafür, dass Liquibase das ChangeSet findet.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <databaseChangeLog
3   xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns:pro="http://www.liquibase.org/xml/ns/pro"
6   xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
```

```
7 http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-4.23.xsd
8 http://www.liquibase.org/xml/ns/pro
9 http://www.liquibase.org/xml/ns/pro/liquibase-pro-4.23.xsd">
10 <includeAll path="db/versions"/>
11 </databaseChangeLog>
```

3. Changelog Konfiguration - output-changelog.xml

Diese Datei enthält sämtliche Datenbankänderungen und die Änderungen werden in Form von ChangeSet beschrieben und verwaltet. Im Folgenden ist ein Abschnitt im XML-Format für das Anlegen einer Tabelle Abonnement wiedergegeben.

```
1 <changeSet author="ng (generated)" id="1743435452264-1">
2   <createTable tableName="abonnement">
3     <column name="id" type="VARCHAR(32)">
4       <constraints nullable="false" primaryKey="true" primaryKeyName="abonnementPK" />
5     </column>
6     <column name="description" type="TEXT" />
7     <column name="label" type="VARCHAR(255)">
8       <constraints nullable="false" />
9     </column>
10    <column name="base_url" type="VARCHAR(255)" />
11    <column name="file_format" type="VARCHAR(255)" />
12    <column name="end_datum" type="date" />
13    <column name="location_id" type="VARCHAR(255)" />
14    <column name="parameter" type="VARCHAR(255)" />
15    <column name="start_datum" type="date" />
16    <column name="sub_url" type="VARCHAR(255)" />
17  </createTable>
18 </changeSet>
```

4. Changelog Konfiguration - liquibase-new-diff-changelog.xml

Diese Datei enthält nur bestimmte Änderungen von bestimmten Tabellen, die man in die Datei `output-changelog.xml` einfügt, wenn man das Schema der Datenbank modifiziert. Ähnlich wie in `output-changelog.xml` werden Änderungen auch in Form von ChangeSet beschrieben.

2.7 Datenbankmanagementsystem

Wie bereits in dem letzten Abschnitt offengelegt, wird ein PostgreSQL als das Datenbankmanagementsystem gewählt. Da alle Datenbanken von Ahu GmbH zentral im lokalen Netzwerk liegen und die meisten Projekte PostgreSQL-Datenbank verwenden, ist es deswegen ersichtlich, dass für dieses Projekt auch eine PostgreSQL-Datenbank verwendet wird. Außerdem spricht der Einsatz einer PostgreSQL-Datenbank aufgrund ihrer zahlreichen Vorteile für sich.

1. Open-Source: geringe Kosten und hohe Flexibilität und Innovation, die bei anderen Datenbanklösungen nicht immer möglich sind.
2. Leistung und Skalierbarkeit : PostgreSQL unterstützt eine Vielzahl von Leistungsoptimierung und verfügt über eine hohe Lese-/Schreibgeschwindigkeit, insbesondere bei der Unterstützung von Geodaten.
3. Fundierte Sprachunterstützung : Aufgrund seiner Kompatibilität und Unterstützung mehrerer Programmiersprachen ist PostgreSQL eine der flexibelsten Datenbanken für Entwickler. Python, JavaScript, C/C++, Java und weitere beliebte Programmiersprachen bieten ausgereifte Unterstützung für PostgreSQL.

3 Anforderungsanalyse und Spezifikation

3.1 Analyse

Während der Analysephase werden die **Stakeholder** identifiziert und deren Anforderungen mittels **User Stories** gesammelt. Die Anforderungen können in drei verschiedenen Untergruppen eingeteilt werden, um die systematisch und spezifisch behandeln zu können.

1. Funktionale Anforderungen beschreiben die Funktionen, die eine ganze Anwendung oder auch nur eine von ihren Komponenten erfüllen soll. Eine Funktion besteht aus drei Schritten: Eingabe der Daten-Systemverhalten-Ausgabe der Daten. Sie kann die Daten berechnen und manipulieren, Geschäftsprozesse ausführen, Benutzerinteraktionen herstellen oder andere Aufgabe ausführen.
2. Nicht-funktionale Anforderungen beschreiben, wie das System es tut, während die funktionalen Anforderungen bestimmen, was das System tut. Darunter zählen die Leistungsstandards und Qualitätsmerkmale von Software, z.B. die Benutzerfreundlichkeit, Effektivität, Sicherheit, Skalierbarkeit usw.
3. Randbedingungen sind Vorgaben, die den Lösungsraum einschränken und das Verhalten der Software beeinflussen.

3.1.1 Anwendungsszenario

Die zu entwickelnde Applikation soll als zentrale Plattform zur Suche, Verwaltung und Bereitstellung von Geodaten dienen. Sie ermöglicht die Aggregation relevanter Informationen aus verschiedenen Datenquellen und stellt diese über eine standardisierte Schnittstelle einem externen Client - wie dem **ahuManager** - zur Verfügung.

3.1.2 Stakeholder und User-Stories

In der Softwareentwicklung bezeichnet der Begriff Stakeholder alle Personen, Gruppen oder Organisationen, die ein Interesse am Erfolg eines Projekts haben und direkt oder indirekt

von dessen Ergebnissen betroffen sind. Um die funktionalen Anforderungen dieser Stakeholder zu erfassen, werden sogenannte User-Stories verwendet. Diese beschreiben aus Sicht der Endnutzer konkrete Anwendungsbedarfe.

User-Stories:

1. Als Benutzer möchte ich in der Web-Oberfläche eine Übersicht über populäre Datenquellen erhalten, die relevante Geodaten bereitstellen.
2. Als Benutzer möchte ich gezielt nach Messdaten wie Grundwasserständen, Pegelständen, Niederschlag, etc. recherchieren und dabei Informationen über das Datenformat und wesentliche Inhalte der Datensätze erhalten.
3. Als Benutzer möchte ich Messdaten, die meist im CSV-Format oder individuell strukturierte ASCII-Dateien vorliegen, automatisiert in ein einheitliches Zielformat konvertieren lassen.
4. Als Benutzer möchte ich die Daten in Form einer Gitter angezeigt bekommen.
5. Als Benutzer möchte ich die Möglichkeit haben, bestimmte Datensätze zu abonnieren.
6. Als Benutzer möchte ich die gefundenen Messdaten visuell auf einer Karte darstellen lassen.

3.2 Spezifikation

Basierend auf den im vorherigen Abschnitt formulierten User-Stories lassen sich die funktionalen Anforderungen der Anwendung konkretisieren. In diesem Zusammenhang werden zentrale **Use-Cases** definiert, welche die Interaktionen zwischen Nutzern und dem System aus funktionaler Sicht beschreiben.

3.2.1 Use-Cases

Ein **Use-Case** beschreibt die sichtbare Interaktion eines Nutzers(Akteurs) mit dem System. Dabei kann es sich beim Akteur um eine Person, eine Organisation oder auch ein externes System handeln. Im Folgenden werden mehrere Use-Cases für die prototypische Anwendung dargestellt.

Use-Case 1: Einsicht historischer Wetterdaten

Akteur : Benutzer

Ziel: Einsicht von Wetterdaten einer bestimmten Messstation

Vorbedingung: keine

Nachbedingung: Die Daten der ausgewählten Station wurden als Diagramm angezeigt

Ablaufbeschreibung:

1. Der Benutzer navigiert zur Ansicht „Historische Wetterdaten“.
2. Die Seite zeigt initial eine leere Karte sowie verschiedene Dropdown-Menüs zur Auswahl der Datenparameter.
3. Der Benutzer wählt eine gewünschte zeitliche Auflösung (z.B. stündlich, täglich, monatlich, jährlich).
4. Daraufhin wird ein Dropdown-Menü mit Hauptparametern (z.B. Temperatur, Niederschlag) eingeblendet
5. Nach Auswahl eines Hauptparameters erscheint ein weiteres Dropdown-Menü mit dazugehörigen Unterparametern.
6. Das System zeigt anschließend eine Deutschlandkarte mit allen Messstationen, die den gewählten Unterparameter liefern.
7. Optional kann der Benutzer die Auswahl weiter eingrenzen – entweder auf ein bestimmtes Bundesland oder auf eine konkrete Messstation.
8. Nach Auswahl einer Station wird diese auf der Karte fokussiert und die zugehörigen Wetterdaten in einem Diagramm visualisiert.

Use-Case 2: Einsicht von Wettervorhersagedaten

Akteur : Benutzer

Ziel: Einsicht von Wettervorhersagedaten einer bestimmten Messstation

Vorbedingung: keine

Nachbedingung: Die Vorhersagedaten wurden als Diagramm angezeigt

Ablaufbeschreibung:

1. Der Benutzer navigiert zur Ansicht „Wettervorhersage“.
2. Die Seite zeigt eine Tabelle mit verschiedenen Parametern zur Anzeige verfügbarer Wetterdaten.

3. Der Benutzer wählt eine Wetterstation aus einem Dropdown-Menü aus.
4. Nach der Auswahl erscheint ein Button zur Überprüfung der Datenverfügbarkeit.
5. Der Benutzer klickt auf den Button und wartet auf eine Rückmeldung des Systems.
6. Das System prüft für alle Parameter in der Tabelle, ob jeweils Daten vorhanden sind.
7. Basierend auf dem Ergebnis ergänzt das System die Tabelle um eine zusätzliche Spalte zur Einsicht verfügbarer Daten.
8. Der Benutzer klickt auf das angezeigte Symbol in der neuen Spalte.
9. Ein Dialogfenster öffnet sich und visualisiert die Vorhersagedaten in Form eines Diagramms.

Use-Case 3: Einsicht von Pegelstanddaten ([PegelOnline](#))

Akteur : Benutzer

Ziel: Einsicht von Pegelstanddaten einer ausgewählten Messstation

Vorbedingung: keine

Nachbedingung: Daten der gewählten Station werden visualisiert.

Ablaufbeschreibung:

1. Der Benutzer navigiert zur Ansicht „Pegelstanddaten“.
2. Auf der linken Seite befinden sich Filteroptionen, auf der rechten Seite eine interaktive Karte mit deutschlandweiten Pegelstationen, farblich codiert nach aktuellem Pegelstatus.
3. Der Benutzer wählt eine Station aus dem Filter „Pegelstation“.
4. Das System zeigt – sofern verfügbar – historische sowie Vorhersagedaten der gewählten Station in einem Diagramm an.

Use-Case 4: Einsicht von Liste der Messstellen eines Klimaphänomen

Akteur : Benutzer

Ziel: Einsicht der Liste der Messstelle eines Klimaphänomen

Vorbedingung: keine

Nachbedingung: Die Messdaten zur gewählten Station und zum Phänomen werden angezeigt

Ablaufbeschreibung:

1. Der Benutzer navigiert zur zentralen Suchansicht.
2. Es wird ein zentrales Eingabefeld angezeigt, in das ein Klimaphänomen eingegeben werden kann (in dieser Entwicklungsphase sind die möglichen Begriffe vorgegeben).
3. Unabhängig vom eingegebenen Begriff erscheint links eine tabellarische Liste relevanter Messstellen aus verschiedenen Datenquellen; rechts wird eine Karte mit zugehörigen Messstellen angezeigt.
4. Der Benutzer filtert die angezeigten Messstationen optional nach Koordinaten oder nach Datenanbieter.
5. In der Tabelle befindet sich eine Spalte mit einem Button „Daten anzeigen“. Der Benutzer klickt darauf.
6. Ein Dialogfenster öffnet sich, in dem ein Dropdown-Menü zur Auswahl spezifischer Datenoptionen erscheint.
7. Nach Auswahl eines Eintrags zeigt das System die tatsächlichen Messdaten zur ausgewählten Station und zum gewählten Klimaphänomen.

Use-Case 5: Daten zu einem Klimaphänomen abonnieren

Akteur : Benutzer

Ziel: Die Daten einer ausgewählten Station zu einem Klimaphänomen abonnieren

Vorbedingung: keine

Nachbedingung: Die Daten einer ausgewählten Station wurde erfolgreich abonniert.

Ablaufbeschreibung:

1. Im geöffneten Dialogfenster befindet sich ein Button „Abonnieren“.
2. Der Benutzer klickt auf den Button, woraufhin sich ein weiteres Dialogfenster öffnet.
3. In diesem Dialog überprüft der Benutzer die Metadaten der zu abonnierenden Daten, kann ein Start- und Enddatum festlegen sowie das gewünschte Datenformat auswählen.
4. Der Benutzer klickt auf den Button „Speichern“. Das Dialogfenster schließt sich automatisch.
5. Anschließend wechselt der Benutzer zur Ansicht „Abonnement-Verwaltung“ und sieht den neu erstellten Abonnement-Eintrag in einer tabellarischen Übersicht.

Use-Case 6: Abonnierte Datensätze in Grid abrufen

Akteur : Benutzer

Ziel: Abruf abonnisierter Datensätze

Vorbedingung: Abonnements sind vorhanden.

Nachbedingung: Die abonnierten Daten wurden erfolgreich angezeigt.

Ablaufbeschreibung:

1. Der Benutzer navigiert zur Ansicht „Abonnement-Verwaltung“.
2. In der rechten Spalte der Grid-Ansicht fährt er mit der Maus über ein Symbol mit dem Tooltip „Abonnierte Daten abrufen“ und klickt drauf.
3. Ein Dialogfenster öffnet sich und bietet zwei Optionen zur Auswahl: „Alle Daten im angegebenen Zeitraum“ und „Inkrementelle Updates“
4. Der Benutzer wählt die Option „Alle Daten im angegebenen Zeitraum“ aus.
5. Das System leitet den Benutzer auf eine separate URL weiter, auf der die abonnierten Datensätze im gewählten Format angezeigt werden.

Use-Case 7: Umformatierung der Rohdaten konfigurieren

Akteur : Benutzer

Ziel: Konfiguration der Umformatierung von Rohdaten

Vorbedingung: Rohdaten sind verfügbar.

Nachbedingung: Die Umformatierungskonfiguration wurde gespeichert.

Ablaufbeschreibung:

1. Der Benutzer klickt in der Grid-Ansicht zur Such-Ansicht auf den Button „Daten anzeigen“.
2. Es öffnet sich ein Dialogfenster, das eine tabellarische Darstellung der Rohdaten sowie die Buttons „Abonnieren“ und „Umformatieren“ enthält. Der Benutzer klickt auf den Button „Umformatieren“.
3. Ein neues Dialogfenster erscheint, in dem der Benutzer Parameter zur Umformatierung der Rohdaten anpassen kann (z.B. Feldtrennung, Strukturierung, Einheiten).
4. Nach der Konfiguration klickt der Benutzer auf „Konfiguration speichern“.
5. Das System speichert die Einstellungen persistent zur späteren Wiederverwendung.

4 Konzeption und Architektur

4.1 Konzeption

Zur strukturierten Beschreibung der Systemarchitektur werden verschiedene Sichten eingesetzt. Diese ermöglichen es, das System aus unterschiedlichen Perspektiven zu betrachten und somit sowohl fachliche als auch technische Anforderungen klar und nachvollziehbar zu adressieren. Im Rahmen dieser Arbeit werden vier wesentliche Architektsichten betrachtet.

1. Kontextsicht

Die Kontextsicht zeigt das zu entwickelnde System als Blockbox, mit dem die Interaktion des Users stattfindet und welche weiteren Systeme beteiligt sind, aus den das zu entwickelnde System Daten importieren oder exportieren kann.

Die Abbildung 4.1 zeigt das System, welche sowohl die Schnittstelle zum Benutzer also auch mit anderen externen Systemen zum Importieren der Daten darstellt. Daraus hinaus ist es mit einem Datenbankmanagementsystem zur Speicherung von Daten verbunden.

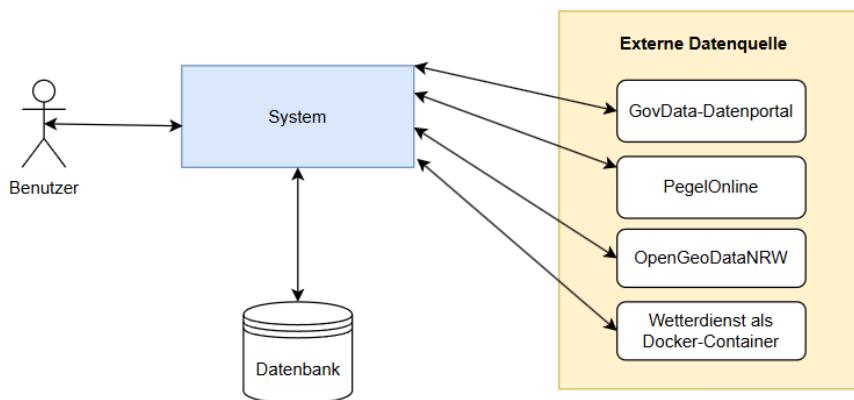


Abbildung 4.1: Kontextangrenzung

2. Bausteinschicht

Die Bausteinschicht zeigt den näheren Aufbau des Systems - und zwar denjenigen Teil, den entwickelt wird. Jedes mit blau gefüllte Package beinhaltet alle Entitäten und die für den Aufbau der View und Datenaustausch mit der externen Datenquelle benötigten Komponenten.

Im Package **MainView** wird die Anwendung mit Startseite initialisiert und die Anbindung der weiteren Views an die Hauptview ermöglicht. Das Package **Utils** beinhaltet zahlreiche Hilfsfunktionen. Diese dienen vor allem dazu, die Modularität, die Wartbarkeit und die Verständlichkeit des Codes sicherzustellen. Das Package **Konfiguration** beinhaltet unterschiedliche Konfigurationsklassen für die Applikation.

Die anderen Packages sind gemäß dem Single Responsibility Principle (SRP) aufgebaut. Jedes Package verfolgt dabei eine klar definierte, spezifische Aufgabe. Durch die Anwendung des SRP werden die Verantwortlichkeiten innerhalb des Systems präzise getrennt, was die Projektstruktur übersichtlicher und verständlicher macht.

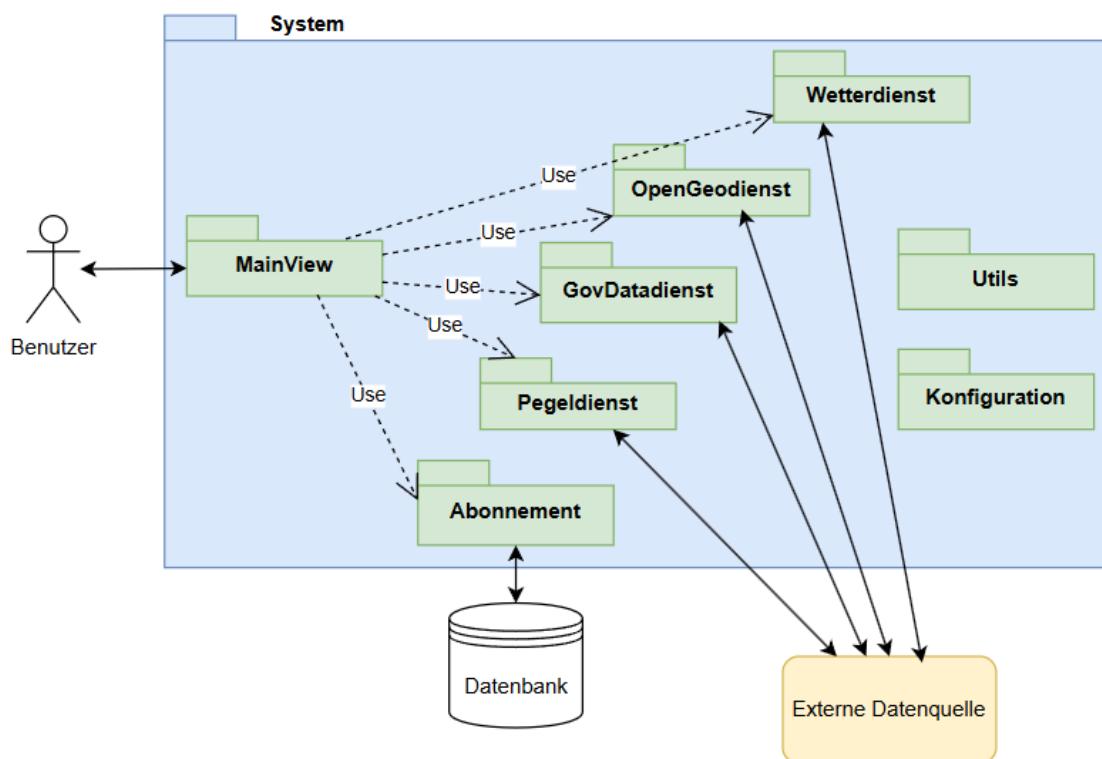


Abbildung 4.2: Bausteinsicht

3. Laufzeitsicht

Die Laufzeitsicht beschreibt, wie ein System während seiner Ausführung seine wesentliche Aufgaben ausführt. Häufig werden Sequenzdiagramme verwendet, um die Laufzeitsicht zu visualisieren, insbesondere zur Darstellung von Interaktionen zwischen Systemteilen.

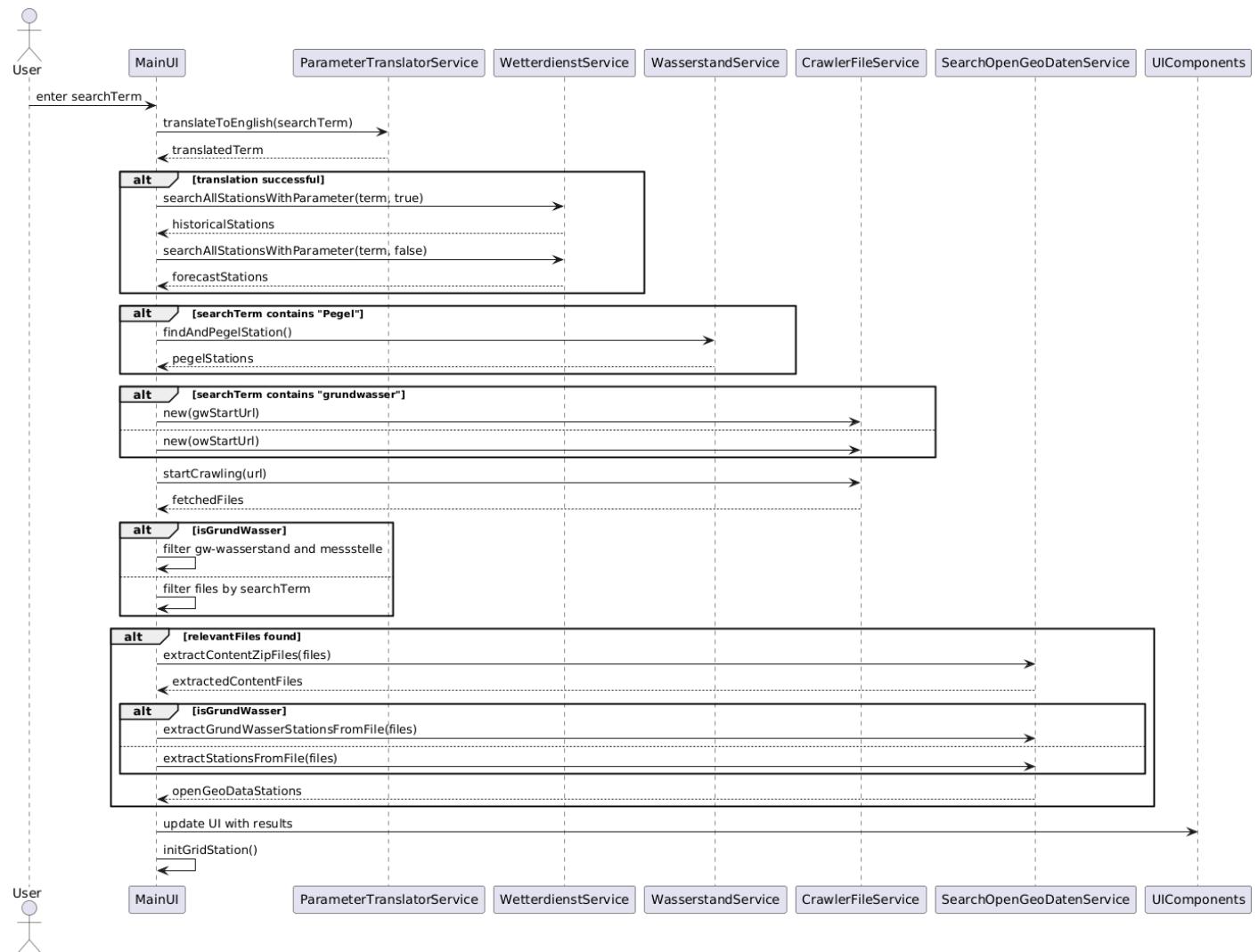


Abbildung 4.3: Use-Case 4: Einsicht der Liste der Messstelle eines Klimaphänomen

Die obige Abbildung demonstriert das Use-Case 4 basierend auf die Nutzer-Eingabe. Ein Benutzer gibt einen Suchbegriff in ein Suchfeld ein. Daraufhin wird:

- Der Begriff ins Englische übersetzt, da die Anfrage an den Server ausschließlich mit englischen Parametern zulässt, um passende Wetterstation über den Deutschen Wetterdienst zu finden (historisch und prognostisch).
- Bei passenden Begriffen wie „Pegel“ oder „Grundwasser“ werden zusätzliche Datenquellen abgefragt:

- Pegelstationen über den Pegeldienst
 - Grundwasser - oder Oberflächendaten über einen Web-Crawler((CrawlerFileService)), der OpenGeoData-Dateien herunterlädt und filtert.
3. Die gefundenen Dateien werden entpackt und analysiert, um Stationsdaten zu extrahieren. Alle Stationen werden zusammengeführt, nach Anbieter gruppiert und dem Benutzer im UI (Gitteransicht + Kartenansicht + Filtermöglichkeiten) dargestellt.

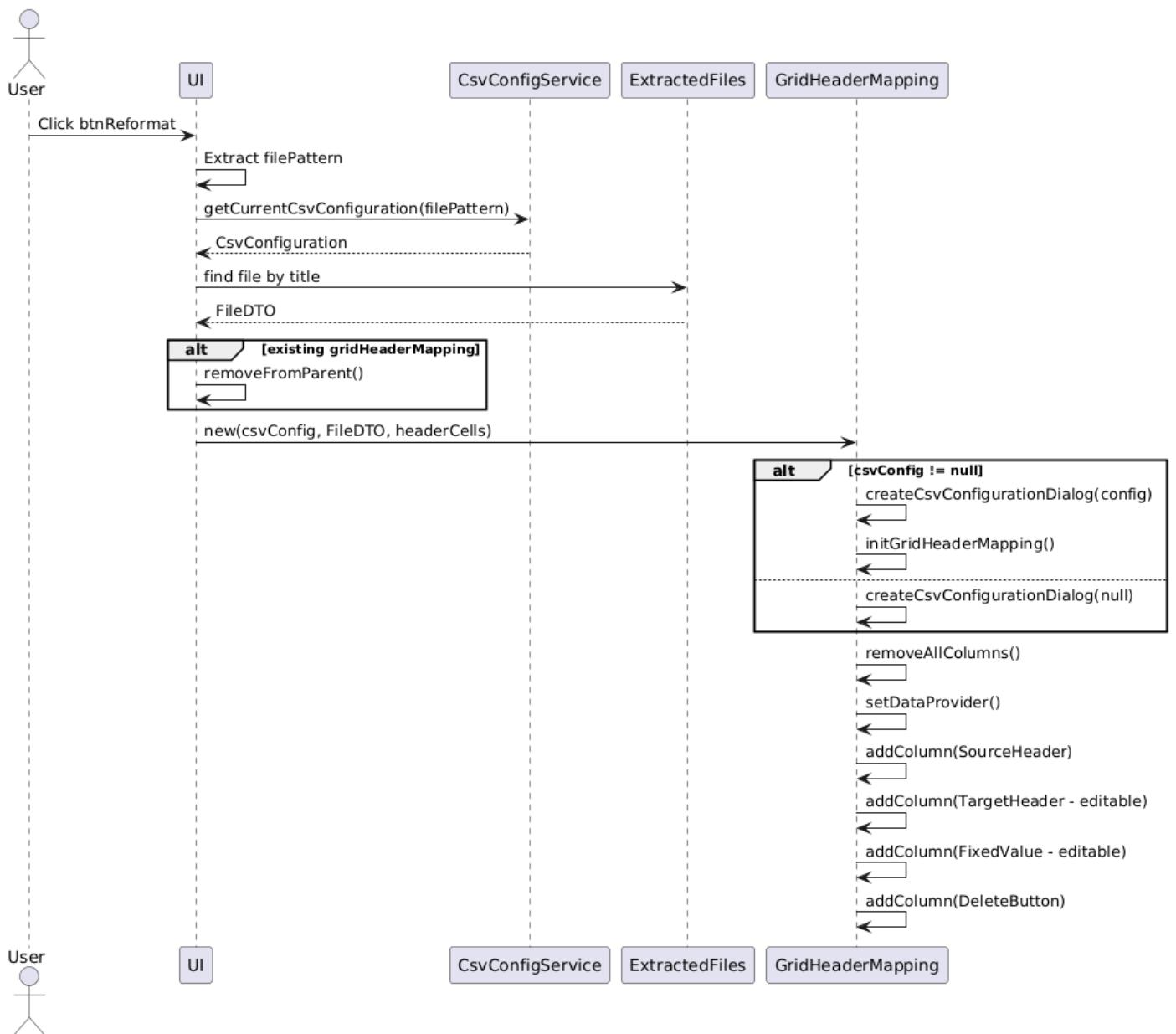


Abbildung 4.4: Use-Case 7: Umformatierung der Rohdaten konfigurieren

Die Abbildung 4.4 illustriert den Fluss vom Use-Case 7. Ein Benutzer wählt eine CSV-Datei aus einer Dropdown-Liste aus und klickt auf „Umformatieren“. Das System:

1. Ermittelt die passende Konfiguration (falls vorhanden) basierend auf dem Dateienamen.
2. Lädt die zugehörige Datei aus bereits extrahierten Dateien.
3. Erzeugt eine neue Zuordnungstabelle (Grid), in der die Spalten der CSV-Datei angezeigt werden.
4. Benutzer kann dann:
 - Quell-und Zielspalten manuell zuordnen.
 - Feste Werte für eine Spalte eintragen.
 - Einträge als Header löschen oder bearbeiten.
5. Diese Konfiguration kann anschließend gespeichert oder für die weitere Verarbeitung verwendet werden.

4. Verteilungsschicht

Üblicherweise beschreibt die Verteilungssicht die technische Infrastruktur und wie sich die einzelnen (Software-)Bausteine auf die einzelnen Infrastrukturkomponenten verteilen.

Im Rahmen dieses Prototyps wird auf die Verteilungsschicht verzichtet, da die entwickelte Anwendung ausschließlich lokal (auf dem *localhost*) ausgeführt wird und kein Deployment auf verteilte Systemumgebungen vorgesehen ist. Eine detaillierte Beschreibung der physischen Infrastruktur ist daher zum aktuellen Zeitpunkt nicht erforderlich.

4.2 Design-Entscheidung

Bei der Architekturarbeit geht es meistens darum, die Entscheidungen zu treffen, die dazu beitragen, konkrete Lösungen für die aktuellen Herausforderungen zu finden. Von daher kommen zunächst zwei Architekturstile bei dieser prototypischen Entwicklung in Frage: **Monolith** und **Event-Driven Architektur**(Ereignisgesteuerte Architektur).

Bei der **monolithischen Architektur** handelt es sich um eine Softwarearchitektur, bei der alle Funktionalitäten und Komponenten einer Anwendung in einem einzigen, zusammenhängenden Codeblock gebündelt sind. Die gesamte Anwendung wird gemeinsame entwickelt, getestet, bereitgestellt und skaliert. Monolithen sind oft einfacher zu konfigurieren und zu verwalten, insbesondere in der frühen Phase eines Projekts.

Im Gegensatz dazu basiert die **Event-Driven Architektur** auf dem Prinzip, dass Komponenten lose gekoppelt sind und über Ereignisse miteinander kommunizieren. Jede Komponente reagiert auf bestimmte Ereignisse und löst ggf. weitere Ereignisse aus. Diese Architektur eignet sich besonders gut für Systeme mit hoher Skalierbarkeit, Flexibilität und dynamischen Anforderungen, ist jedoch oft komplexer in der Aufstellung der Infrastruktur und im Betrieb.

Für die prototypische Entwicklung habe ich mich für die **monolithische Architektur** entschieden, da sie einen schnelleren Entwicklungsstart ermöglicht, weniger Infrastruktur erfordert und die Komplexität im Vergleich zur **Event-Driven-Architektur** zunächst gering ist. Diese Entscheidung erlaubt es, mehr den Fokus auf die Umsetzungen der Kernfunktionalitäten zu legen, ohne sich frühzeitig mit verteilten Systemen oder asynchroner Kommunikation auseinandersetzen zu müssen.

5 Implementierung

In diesem Kapitel wird über die Implementierung einzelner wichtiger Komponenten für die zuvor gewählte Architektur berichtet und ebenfalls auf während der Umsetzung auftretende Schwierigkeiten eingegangen.

5.1 Übersicht über wichtige Klassen und Entitäten

Wie bereits am Anfang des 2.Kapitels erwähnt, wird als Einstiegspunkt für die Entwicklung eine Vaadin Applikation durch Bereitstellung von Vaadin Initializer genommen. Dies Package enthält die grundlegenden Komponente für eine simple Benutzeroberfläche.

In diesem Package gibt es eine kurz Main-Klasse(**Application.java**) als Startpunkt für die Applikation, die folgenden Aufbau hat:

```
1 @SpringBootApplication
2 @Theme(value = "opendata-konnektor")
3 @EnableCaching
4 public class Application implements AppShellConfigurator {
5     public static void main(String[] args) {
6         SpringApplication.run(Application.class, args);
7     }
8 }
```

Die Annotation **@SpringBootApplication** kombiniert aus:

1. **@Configuration**: markiert die Klasse als Konfigurationsklasse für Spring.
2. **@EnableAutoConfiguration**: ermöglicht die automatische Konfiguration basierend auf den Abhängigkeiten im Klassenpfad.
3. **@ComponentScan**: sucht nach Komponenten, Konfigurationsklassen und Services im aktuellen Paket und allen Unterpaketen.

Die Annotation **@Theme(value = "opendata-konnektor")** legt das Vaadin-Theme fest. **value** definiert den Namen des Themes und Themens werden verwendet, um das Erscheinungsbild der Anwendung anzupassen.

Die Annotation **@EnableCaching** aktiviert die Spring Caching-Funktionalität und ermöglicht die Verwendung von Cache-Mechanismen(**@Cacheable**,**@Cacheable**,**@CacheEvict**).

AppShellConfigurator ist eine Schnittstelle (**Interface**) aus dem Vaadin Framework. Sie wird verwendet, um die **App Shell** einer Vaadin-Anwendung zu konfigurieren. Was ist die App Shell in Vaadin ?

- Die App Shell ist ein grundlegendes HTML-Dokument, das vom Server an den Client gesendet wird.
- Es enthält die wesentlichen Informationen wie Meta-Tags, Icons, Stylesheets usw., die bei der ersten Anfrage geladen werden.
- Die App Shell bleibt auch unverändert, während der Inhalt dynamisch aktualisiert wird.

Um jedes Object oder jede Entität möglichst spezifisch zu halten, wird am Anfang eine grundlegende, nicht abgeleitete Basisentität, die als Vorlage oder Basis dient, auf der andere Entitäten aufgebaut werden können, indem sie deren Eigenschaften erben, ohne diese erneut zu definieren. In Java-Programmiersprache wird dieses Verhalten als **Vererbung** bezeichnet. Als Beispiel wird die Klasse „ **BaseEntity**“ erläutert:

```
1 @MappedSuperclass
2 @Getter
3 @Setter
4 public class BaseEntity implements Comparable<BaseEntity> {
5     @Id
6     @GeneratedValue(generator = "generateIfNotAssigned")
7     @GenericGenerator(name = "generateIfNotAssigned", strategy =
8         "org.hibernate.id.UUIDHexGenerator")
9     @Column(length = 32)
10    @Access(jakarta.persistence.AccessType.PROPERTY)
11    private String id;
12
13    @Length(min = 1)
14    @NotNull
15    private String label;
16
17    @Column(columnDefinition = "TEXT")
18    private String description;
19
20    @Column(name = "base_url")
21    private String url;
```

```
22 @Column(name = "last_updated")
23 private LocalDate lastUpdated;
24
25 public int compareTo(BaseEntity o) {
26     return (id != null) ? id.compareTo(o.getId()) : hashCode() - o.hashCode();
27 }
28 @Override
29 public int hashCode() {
30     return (id != null) ? id.hashCode() : 1;
31 }
32 @Override
33 public boolean equals(Object obj) {
34     if (!(obj instanceof BaseEntity)) {
35         return false;
36     }
37     BaseEntity be = (BaseEntity) obj;
38     return (id != null) ? id.equals(be.getId()) : this == be;
39 }
40 }
```

Die Annotation **@MappedSuperclass** kennzeichnet die Klasse als Basisklasse für andere JPA-Entitäten. Die Felder und Methoden dieser Klasse werden nicht direkt in der Datenbank abgebildet, sondern in die Tabellen der abgeleiteten Klassen integriert. Außerdem wird es verwendet, um gemeinsame Eigenschaften (**id**, **label**), etc. zentral zu definieren.

Die Annotation **@Getter** und **@Setter** sind Lombok Annotation, die automatisch Getter/Setter-Methoden für alle Felder der Klasse generiert.

Die Annotation **@Id** ist ein JPA-Annotation und kennzeichnet das Primärschlüsselfeld der Entität.

Die Annotation **@GeneratedValue(generator = "generateIfNotAssigned")** ist ebenfalls ein JPA-Annotation und gibt an, dass der Wert für **id** automatisch generiert wird. Die Option **generator** verweist auf einen benutzerdefinierten Generator.

Die Annotation **@GenericGenerator**) ist eine Hibernate-spezifische Annotation und definiert einen benutzerdefinierten ID-Generator. Die Option **name** ist der Name des Generators, hier „generateIfNotAssigned“. Die Option **strategy** ist die Strategie für die ID-Erzeugung, und **UUIDHexGenerator** generiert UUIDs als Hex-Strings(32 Zeichen).

Die Annotation **@Access(jakarta.persistence.AccessType.PROPERTY)** definiert, wie auf Felder/Methoden zugegriffen wird. In diesem Fall erfolgt der Zugriff mit **PROPERTY** über

Getter und Setter-Methoden.

Die Annotation `@Length(min = 1)` ist eine Hibernate-Validator Annotation und legt die minimale Länge für das Feld auf 1 Zeichen fest.

Die Annotation `@NotNull` ist ein Bean Validation Annotation und verhindert, dass `label` auf `null` gesetzt wird und immer einen Wert haben muss.

Die Annotation `@Column(columnDefinition = "TEXT")` ist JPA Annotation und definiert die Datenbankspalte `description` als `TEXT` und `TEXT` erlaubt lange Zeichenketten (mehr als `VARCHAR`).

```
1  implements Comparable<BaseEntity>
```

Damit kann die Klasse verglichen und sortiert werden, basierend auf der `id` in der Methode `compareTo`. Die Annotation `@Override` stellt sicher, dass die Methode die korrekte Signatur aus dem `Comparable` Interface hat.

Um die Verwendung der obigen beschriebenen Klasse zu veranschaulichen, wird als Beispiel die Klasse „**Abonnement**“ erläutert:

```
1  @Entity
2  @Table(name = "abonnement")
3  @Getter
4  @Setter
5  @NoArgsConstructor
6  @AllArgsConstructor
7  public class Abonnement extends BaseEntity {
8      @Column(name = "file_format")
9      private String dateiFormat;
10     private String parameter;
11     @Column(name = "location_id")
12     private String locationId;
13     @Column(name = "start_datum")
14     private LocalDate startDatum;
15     @Column(name = "end_datum")
16     private LocalDate endDatum;
17     @Column(name = "sub_url")
18     private String subscriptionUrl;
19 }
```

Die Klasse **Abonnement** erbt von der Basisklasse **BaseEntity**. Dadurch übernimmt sie automatisch alle Eigenschaften (`id, label, description, url, etc.`) und Methoden (`compareTo, hashCode, equals`) der Basisklasse.

Die Annotation `@Entity` markiert die Klasse als JPA-Entität und wird in der Datenbank als Tabelle gespeichert. Die Annotation `@NoArgsConstructor` (Lombok) generiert einen parameterlosen Konstruktor und `@AllArgsConstructor` hingegen einen Konstruktor mit allen Feldern als Parameter.

Anschließend an die Entität, wird das zugehörige Repository, in Form eines Interfaces implementiert. Es erweitert `JpaRepository`, wodurch eine Vielzahl von CRUD-Operationen und zusätzlichen Methoden bereitgestellt werden.

```
1 @Repository  
2 public interface AbonnementRepository extends JpaRepository<Abonnement, String>{}
```

Die Annotation `@Repository` kennzeichnet die Schnittstelle als Datenbank-Repository und wird von Spring verwendet, um die Klasse als Datenzugriffsschicht zu erkennen und ermöglicht die automatische Fehlerbehandlung bei Datenbankoperationen.

`JpaRepository<T, ID>` ist eine generische Schnittstelle, die CRUD-Methoden bereitstellt:

- T : die Entitätsklasse, die verwaltet wird, in diesem Fall(Abonnement).
- Id: der Datentyp des Primärschlüssels (String).

Für die Entität **Abonnement** wird ein sogenannten AbonnementController bereitgestellt, mit dem der Zugriff auf die Daten von außen ermöglicht.

```
1 @RestController  
2 @RequestMapping("/api/subscription")  
3 public class AbonnementController {  
4     @Autowired  
5     private AbonnementService abonnementService;  
6  
7     @GetMapping(value = "/data")  
8     public ResponseEntity<String> getAboData(@RequestParam @NotBlank String id) {  
9         return abonnementService.fetchAbonnementData(id).orElseGet(() ->  
10            ResponseEntity.status(HttpStatus.NOT_FOUND).body("kein Abonnement fuer  
11            die " + id + " gefunden."))  
12        }  
13        @GetMapping(value = "/incremental_data")  
14        public ResponseEntity<String> getIncrementalData(@RequestParam @NotBlank String  
15            id) {  
16            return abonnementService.fetchInrementalData(id).orElseGet(  
17                () -> ResponseEntity.status(HttpStatus.NOT_FOUND).body("Kein Abonnement fuer  
18                die " + id + " gefunden."));  
19        }  
20    }
```

Die Annotation **@RestController** ist eine Kombination aus **@Controller** und **@ResponseBody** und markiert diese Klasse als REST-Controlle und die Methodenrückgaben werden automatisch in HTTP-Responses konvertiert.

Die Annotation **@RequestMapping(/api/subscription)** definiert den Basis-URL-Pfad für alle Endpunkte in dieser Klasse. Mit der Annotation **@Autowired** sucht Spring automatisch nach einer passenden Bean und injiziert diese in das Controller-Feld **abonnementService**. Dies ist in Spring als **Dependency Injection** genannt.

Die Annotation **@GetMapping(/data)** definiert einen GET-Endpunkt unter /api/subscription/data und **@RequestParam** extrahiert den **id**-Parameter aus der URL-Abfragezeichenkette. **@NotBlank** ist ein Bean Validation und validiert, dass der **id**-Parameter nicht leer ist und falls der leer ist, wird ein 400 Bad Request zurückgegeben. **ResponseEntity<String>** wird verwendet, um eine HTTP-Response mit einem StatusCode und einem Body zurückzugeben.

5.2 Front-End-Komponente

Im Rahmen dieses Abschnittes wird die Implementierung des Use-Cases 4: Einsicht von Liste der Messstellen eines Klimaphänomen erläutert. Die Entwicklung erfolgt unter Verwendung von Vaadin-Framework.

Die Umsetzung eines UI-Komponents erfolgt unter Java-Programmiersprache als Klasse, welche ein Komponent wie z.B. ein VerticalLayout oder HorizontalLayout oder ein Dialog von Paket **com.vaadin.flow.component.Component** erbt.

```
1 import com.vaadin.flow.component.combobox.ComboBox;
2 import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
3 import com.vaadin.flow.component.orderedlayout.VerticalLayout;
4
5 @Route(value = "suche", layout = MainLayout.class)
6 @Slf4j
7 public class SearchGeoDataView extends VerticalLayout {
8
9     private ComboBox<String> tfSearch = new ComboBox<>();
10    tfSearch.setPlaceholder("Suche nach Phänonomen");
11    tfSearch.setClearButtonVisible(true);
12    List<String> phaenomenList = List.of("Niederschlag", "Grundwasser", "Pegel",
13        "Bodentemperatur",
```

```
13 "Lufttemperatur", "Luftfeuchtigkeit", "Windgeschwindigkeit", "Windrichtung",
14     "Windstärke", "Windböe",
15 "Sonnenscheindauer", "Gesamtbewölkung");
16 tfSearch.setItems(phaenomenList.stream().sorted().toList());
17 tfSearch.setAllowCustomValue(true);
18 tfSearch.setWidth("20%");
19 tfSearch.getStyle().set("borderRadius", "12px").set("padding", "0.5rem 1rem")
20 .set("boxShadow", "0 2px 6px rgba(0, 0, 0, 0.1)").set("backgroundColor",
21 "#fff");
22
23 HorizontalLayout searchLayout = new HorizontalLayout(tfSearch);
24 searchLayout.setWidthFull();
25 searchLayout.setJustifyContentMode(JustifyContentMode.CENTER);
26 add(searchLayout);
27 }
```

Mit dem obigen Code ergibt sich folgende Abbildung ohne die Navigationsleiste:

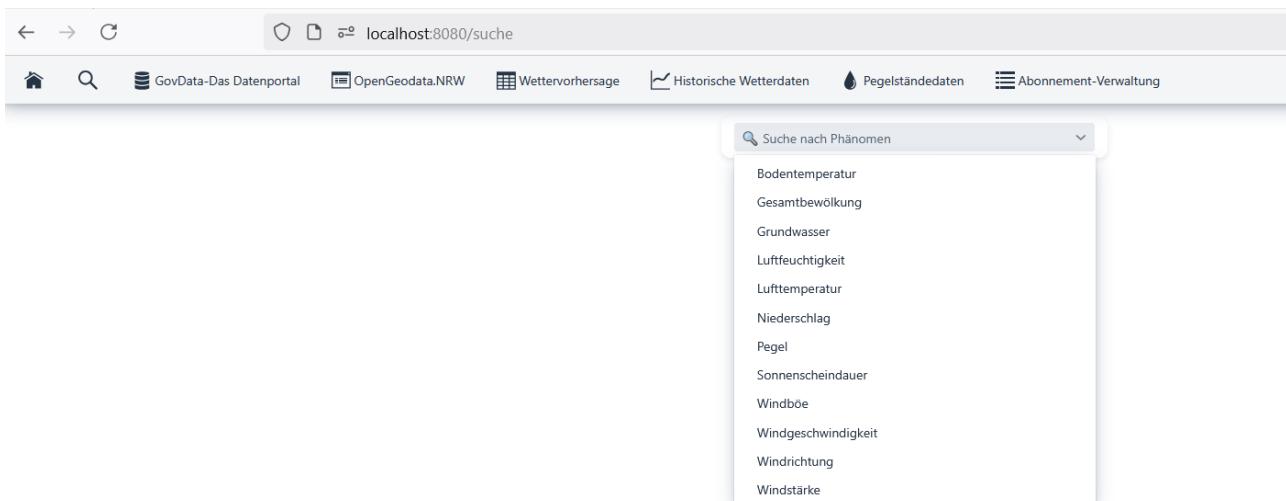


Abbildung 5.1: Suche Seite

Wie üblich in Java, initialisiert man ein Komponente in Vaadin auch mit dem Schlüsselwort **new**. Bei dem Beispiel wird ein DropDown-Feld zur Auswahl von **String**-Werten erstellt.

Mit Abruf von **tfSearch.getStyle()** wird dies DropDown stilisiert und mit **tfSearch.setItems(...)** mit vordefinierten Phänomenen gefüllt. Mit **HorizontalLayout searchLayout = new HorizontalLayout(tfSearch)** wird ComboBox in ein **HorizontalLayout** eingebettet und zentriert ausgerichtet.

Am Ende mit **add(searchLayout)** wird das Layout zur Benutzeroberfläche hinzugefügt.

Zurück zu der Navigationsleiste, um so einen Leiste in Vaadin erstellen zu können, wird anhand des Code ins Details eingegangen.

Die Annotation **@Layout** wird verwendet, um die Klasse als Layout festzulegen. In Vaadin bestimmt ein Layout, wie der Seiteninhalt strukturiert wird, insbesondere wie Komponenten innerhalb des Layouts angeordnet werden.

Die Annotation **@AnonymousAllowed** erlaubt es anonymen Benutzern (nicht authentifizierten Benutzern), auf dieses Layout zuzugreifen. Ohne diese Annotation wäre der Zugriff auf diese Klasse möglicherweise eingeschränkt.

```
1 @Layout
2 @AnonymousAllowed
3 public class MainLayout extends AppLayout {
4     public MainLayout() {
5         addToNavbar(createHeaderContent());
6     }
7     private Component createHeaderContent() {
8         Header header = new Header();
9         header.addclassNames(BoxSizing.CONTENT, Display.FLEX, FlexDirection.COLUMN,
10                           Width.FULL);
11
12         Nav navMain = new Nav();
13         navMain.addclassNames(Display.FLEX, Overflow.AUTO, Padding.Horizontal.MEDIUM,
14                               Padding.Vertical.XSMALL,
15                               TextColor.PRIMARY, BoxShadow.MEDIUM, Width.FULL);
16         UnorderedList navMainItemList = new UnorderedList();
17         navMainItemList.addclassNames(Display.INLINE_FLEX, Gap.LARGE,
18                                       ListStyleType.NONE, Margin.NONE, Padding.SMALL,
19                                       TextColor.PRIMARY);
20         navMain.add(navMainItemList);
21
22         for (MenuItemInfo menuItem : createMainMenuItems()) {
23             navMainItemList.add(menuItem);
24         }
25
26         HorizontalLayout navControls = new HorizontalLayout();
27         navControls.setSpacing(false);
28         navControls.setJustifyContentMode(FlexComponent.JustifyContentMode.BETWEEN);
29         navControls.add(navMain);
30         header.add(navControls);
```

```
28     return header;
29 }
30
31 private MenuItemInfo[] createMainMenuItems() {
32     return new MenuItemInfo[] { new MenuItemInfo(null, VaadinIcon.HOME.create(),
33         StartView.class),
34         new MenuItemInfo(null, VaadinIcon.SEARCH.create(),
35             SearchGeoDataView.class),
36         new MenuItemInfo("GovData-Das Datenportal", VaadinIcon.DATABASE.create(),
37             GridGovData.class),
38         new MenuItemInfo("OpenGeodata.NRW", VaadinIcon.MODAL_LIST.create(),
39             OpenDataNrwView.class),
40         new MenuItemInfo("Wettervorhersage", VaadinIcon.TABLE.create(),
41             WettervorhersageView.class),
42         new MenuItemInfo("Historische Wetterdaten",
43             VaadinIcon.SPLINE_CHART.create(),
44             HistoricalWetterDatenView.class),
45         new MenuItemInfo("Pegelständedaten", VaadinIcon.DROP.create(),
46             PegelstandView.class),
47         new MenuItemInfo("Abonnement-Verwaltung", VaadinIcon.LINES_LIST.create(),
48             GridAbonnement.class) };
49 }
50
51 public static class MenuItemInfo extends ListItem {
52     private final Class<? extends Component> view;
53
54     public MenuItemInfo(String menuTitle, Component icon, Class<? extends
55         Component> view) {
56         this.view = view;
57         RouterLink link = new RouterLink();
58         link.addClassNames(Display.FLEX, Gap.XSMALL, Height.MEDIUM,
59             AlignItems.CENTER, Padding.Horizontal.SMALL,
60             TextColor.BODY);
61         link.setRoute(view);
62         Span text = new Span(menuTitle);
63         text.addClassNames(FontWeight.MEDIUM, FontSize.MEDIUM, Whitespace.NOWRAP);
64
65         if (icon != null) {
66             link.add(icon);
67         }
68         link.add(text);
69         add(link);
70     }
71 }
```

```
59     }
60     public Class<?> getView() {
61         return view;
62     }
63 }
64 }
```

Die Klasse **MainLayout** erweitert **AppLayout**, welches in Vaadin ein standardisiertes Layout ist, das eine Navigationsleiste und einen Inhaltsbereich definiert. Im Konstruktor wird **createHeaderContent()** aufgerufen und zum Navigationsbereich (**Navbar**) hinzufügt.

Die Methode **createHeaderContent()** erstellt den Header-Bereich der Anwendung. Die Komponente **Header** wird erstellt und gestylt. Das Navigationselement mit **Nav** wird erstellt und eine **UnorderedList** wird hinzugefügt. Für jedes **MenuItemInfo** wird ein Listenelement **ListItem** hinzufügt. Der Header wird zurückgegeben und in die Navbar eingefügt.

Die Methode **createMainMenuItems()** erzeugt ein Array von **MenuItemInfo**- Objekten. Jedes **MenuItemInfo**- Objekte repräsentiert einen Menüpunkt mit einem Titel, einem Icon und einer zugehörigen View-Klasse.

Die innere Klasse **MenuItemInfo** erweitert **ListItem** und stellt ein Navigationsmenüelement dar. Konstruktor erstellt einen **RouterLink**, fügt optional Icon und den Text hinzu. Die Methode **getView()** gibt die View-Klasse zurück, die beim Klicken auf das Menüelement aufgerufen wird.

Kapitel 5. Implementierung

Wählt man nun einen Eintrag aus der DropDownList aus,z.B.**Pegel**, ergibt sich folgende Abbildung:

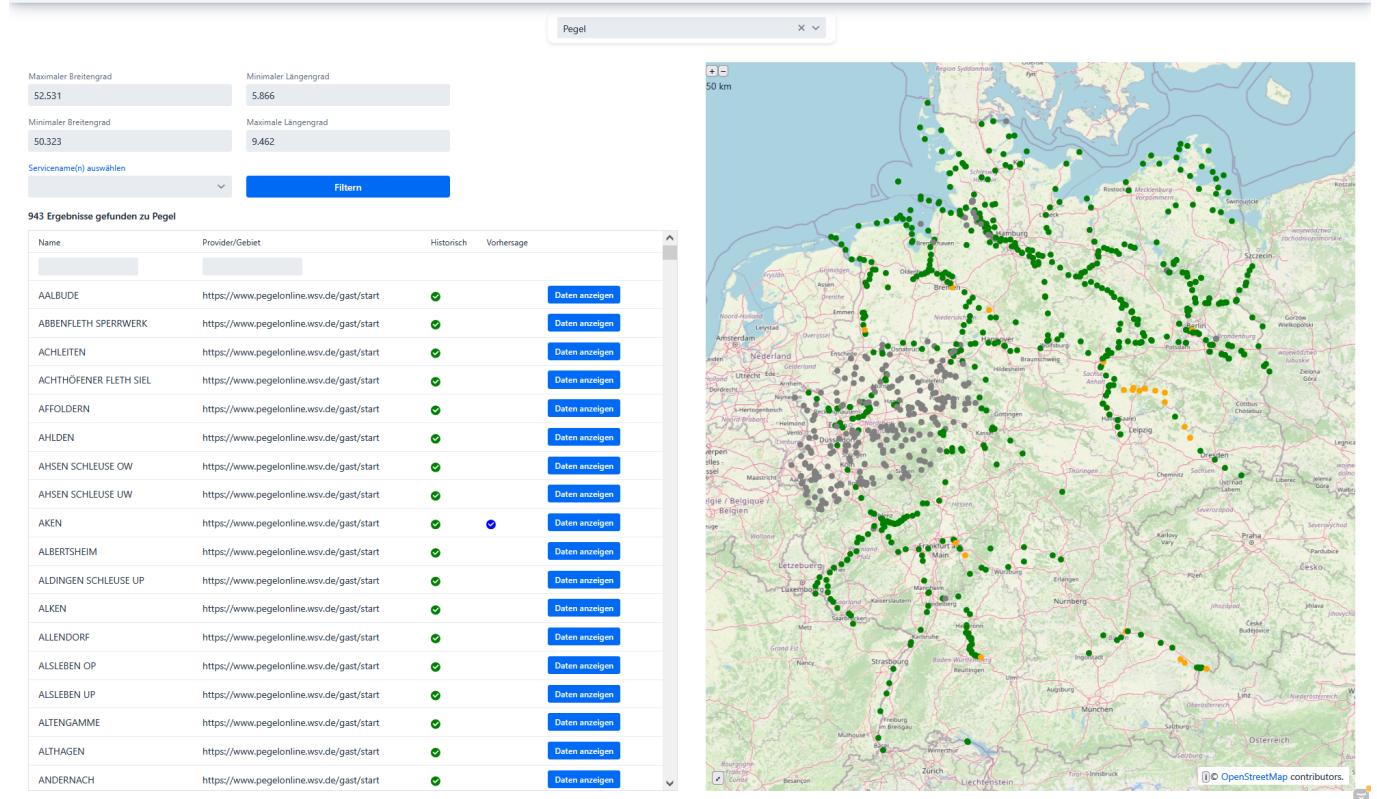


Abbildung 5.2: Einsicht von Liste der Pegelstation

Bei der Abbildung sind es drei Hauptkomponenten deutlich zu erkennen, nämlich : die Tabelle, die Karte und die Filtermöglichkeiten nach Koordinaten und Servicename.

1. Die Tabelle

In Vaadin wird eine Tabelle als Grid genannt und wird in dem Code wie folgt initialisiert.

```
1 private Grid<StationDTO> stationGrid = null;  
2  
3 private Grid.Column<StationDTO> nameColumn = null;  
4 private Grid.Column<StationDTO> providerColumn = null;  
5 private Grid.Column<StationDTO> historischColumn = null;  
6 private Grid.Column<StationDTO> vorhersageColumn = null;  
7 private Grid.Column<StationDTO> btnColumn = null;  
8 private StationDTOFilter filter;  
9 private HeaderRow headerRow = null;  
10 private StationDTO currentStation = null;
```

```
11
12 private void initGridStation() {
13     stationGrid = new Grid<>(StationDTO.class, false);
14     dataProviderStations = stationGrid.setItems(mergedStations);
15
16     nameColumn = stationGrid.addColumn(station ->
17         station.getName()).setHeader("Name")
18         .setAutoWidth(true).setFlexGrow(1);
19     providerColumn = stationGrid.addColumn(station ->
20         station.getProvider()).setHeader("Provider/Gebiet")
21         .setAutoWidth(true).setResizable(true).setFlexGrow(1);
22     historischColumn = stationGrid.addColumn(new ComponentRenderer<>(station -> {
23         if (station instanceof WetterStationDTO wetter) {
24             Icon statusIndicator = new Icon();
25             if (wetter.getIsHistorical() != null && wetter.getIsHistorical()) {
26                 statusIndicator = VaadinIcon.CHECK_CIRCLE.create();
27                 statusIndicator.setColor("green");
28                 statusIndicator.addClassName("text-body");
29                 statusIndicator.setSize("1em");
30             }
31             return statusIndicator;
32         }
33         if (station instanceof PegelStationDTO pegel) {
34             Icon statusIndicator = new Icon();
35             if (pegel.getIsHistorical() != null && pegel.getIsHistorical()) {
36                 statusIndicator = VaadinIcon.CHECK_CIRCLE.create();
37                 statusIndicator.setColor("green");
38                 statusIndicator.addClassName("text-body");
39                 statusIndicator.setSize("1em");
40             }
41             return statusIndicator;
42         }
43         return null;
44     })).setHeader("Historisch").setAutoWidth(true).setFlexGrow(0);
45
46     vorhersageColumn = stationGrid.addColumn(new ComponentRenderer<>(station -> {
47         if (station instanceof WetterStationDTO wetter) {
48             Icon statusIndicator = new Icon();
49             if (wetter.getIsForecast() != null && wetter.getIsForecast()) {
```

```
50     statusIndicator.addClassName("text-body");
51     statusIndicator.setSize("1em");
52 }
53     return statusIndicator;
54 }
55 if (station instanceof PegelStationDTO pegel) {
56     Icon statusIndicator = new Icon();
57     if (pegel.getIsForecast() != null && pegel.getIsForecast()) {
58         statusIndicator = VaadinIcon.CHECK_CIRCLE.create();
59         statusIndicator.setColor("blue");
60         statusIndicator.addClassName("text-body");
61         statusIndicator.setSize("1em");
62     }
63     return statusIndicator;
64 }
65 return null;
66 }).setHeader("Vorhersage").setAutoWidth(true).setFlexGrow(0);
67
68 btnColumn = stationGrid.addColumn(new ComponentRenderer<>(st -> {
69     Button btnShowData = new Button("Daten anzeigen");
70     btnShowData.addThemeVariants(ButtonVariant.LUMO_SMALL,
71         ButtonVariant.LUMO_PRIMARY);
72
73     btnShowData.addClickListener(click -> {
74         if (dialogData == null) {
75             dialogData = new Dialog();
76         } else {
77             dialogData.removeAll();
78             dialogData = new Dialog();
79         }
80         ....
81         // Weitere Logik fuer die Datenverarbeitung
82     });
83     return btnShowData;
84 }).setAutoWidth(true).setFlexGrow(1);
85
86 stationGrid.setColumnOrder(nameColumn, providerColumn, historischColumn,
87     vorhersageColumn, btnColumn);
88 stationGrid.setSizeFull();
89
90 stationGrid.getHeaderRows().clear();
```

```
89 headerRow = stationGrid.appendHeaderRow();
90 filter = new StationDTOFilter(dataProviderStations);
91 headerRow.getCell(nameColumn).setComponent(createFilterHeader(filter::setStationName));
92 headerRow.getCell(providerColumn).setComponent(createFilterHeader(filter::setProvider));
```

Bei dem Code-Abschnitt wird ein **Grid<StationDTO>** initialisiert und es erstellt fünf Spalten:

1. **Name** – Zeigt den Namen der Station an.
2. **Provider/Gebiet** – Zeigt den Anbieter oder das Gebiet der Station an.
3. **Historisch** – Zeigt ein grünes Icon an, wenn die Station historisch ist (`getIsHistorical()` ist true)
4. **Vorhersage** – Zeigt ein blaues Icon an, wenn Vorhersagedaten verfügbar sind (`getIsForecast()` ist true).
5. **Daten anzeigen** – Fügt einen Button hinzu, der beim Klicken ein Dialogfenster zur Datenanzeige öffnet.
6. Zusätzlich wird ein HeaderRow für die Filterung hinzugefügt, der Filter für Name und Provider ermöglicht.

2. Die Karte

Die Karte ist eine Komponente, die an vielen Stellen in der Applikation wiederverwendbar ist, deswegen ist es sinnvoll, diese als eine eigene Klasse auszulagern.

Die Methode zum Aufruf der Karte-Komponente:

```
1 private void initOpenLayersMap() {
2     if (!mapVerticalLayout.getChildren().anyMatch(component -> component instanceof
3         OpenLayersMap)) {
4         olMap = new OpenLayersMap();
5         olMap.setSizeFull();
6         olMap.showStationsOnMap(dataProviderStations.getItems()
7             .collect(Collectors.toList()));
8         mapVerticalLayout.add(olMap);
9     } else {
10         olMap.showStationsOnMap(dataProviderStations.getItems()
11             .collect(Collectors.toList()));
12     }
13 }
```

Die Methode `initOpenLayersMap()` initialisiert eine `OpenLayersMap`-Komponente in einem Vaadin-Layout :

- prüft, ob bereits eine `OpenLayersMap` im `mapVerticalLayout` vorhanden ist.
- Falls nicht vorhanden, wird eine neue `OpenLayersMap` erstellt und den Stationen aus dem `dataProviderStations`-Datensatz hinzugefügt.
- Falls bereits vorhanden, wird die bestehende Karte aktualisiert, um die Stationen anzuzeigen.
- Ein **DataProvider** in Vaadin ist eine Datenquelle, die Daten für UI-Komponenten wie Grids bereitstellt. Es verwaltet die Daten und stellt Methoden zur Verfügung, um Daten zu filtern, zu sortieren und zu aktualisieren.

```
1 @NpmPackage(value = "ol", version = "8.2.0")
2 @NpmPackage(value = "ol-ext", version = "4.0.13")
3 @NpmPackage(value = "ol-layerswitcher", version = "4.1.1")
4 @NpmPackage(value = "ol-popup", version = "5.1.0")
5 @Tag("openlayers")
6 @JsModule("./src/openlayers-connector.js")
7 @Slf4j
8 public class OpenLayersMap extends Div {
9     public OpenLayersMap() {
10         this.getParent().ifPresent(null);
11         initConnector();
12         addDetachListener(detach -> { shutDown();});
13     }
14     public <T extends RestDTO> void showStationsOnMap(List<T> stations) {
15         runBeforeClientResponse(ui -> {
16             boolean isWetterstation = false;
17             if (stations != null && !stations.isEmpty()) {
18                 isWetterstation = stations.get(0) instanceof WetterStationDTO;
19             }
20             String jsArray = buildStationsJson(stations);
21             ui.getPage()
22                 .executeJs("window.Vaadin.Flow.openLayersConnector.showStations($0, $1,
23                             $2)", getElement(),
24                             jsArray, isWetterstation);
25         });
26         private <T extends RestDTO> String buildStationsJson(List<T> stations) {
27             try {
```

```
28     if (stations == null || stations.isEmpty()) {
29         return "[ ]";
30     }
31     return objectMapper.writeValueAsString(stations);
32 } catch (Exception e) {
33     e.printStackTrace();
34     return "[ ]";
35 }
36 }
37 private void initConnector() {
38     runBeforeClientResponse(ui -> ui.getPage()
39         .executeJs("window.Vaadin.Flow.openLayersConnector.initLazy($0)",
40             getElement()));
41 }
```

Die Klasse **OpenLayersMap** erweitert Div und integriert eine **OpenLayers**-Kartenkomponente.
Annotationen:

- **@NpmPackage**: importiert JavaScript-Pakete (ol, ol-ext, ol-layerswitcher, ol-popup) in der angegebenen Version.
- **@Tag**: definiert das HTML-Tag als **openlayers**.
- **@JsModule**: verweist auf die JavaScript-Datei **openlayers-connector.js** zur Client-Integration.

Methoden:

- **showStationsOnMap(List<T> stations)**: zeigt Stationen auf der Karte an und erkennt, ob es sich um Wetterstationen handelt.
- **buildStationsJson(List<T> stations)**: konvertiert die Stationen-Liste in ein JSON-Array für die Client-Seite.
- **initConnector()**: Initialisiert den JavaScript-Connector(initLazy).

3. Die Filtermöglichkeiten

```
1 private FormLayout formLayout = new FormLayout();
2 private TextField tfMinLatitude = new TextField("Minimaler Breitengrad");
3 private TextField tfMaxLatitude = new TextField("Maximaler Breitengrad");
4 private TextField tfMinLongitude = new TextField("Minimaler Längengrad");
5 private TextField tfMaxLongitude = new TextField("Maximale Längengrad");
```

```
6 private Button btnFilterStations = new Button("Filtern");
7
8 tfMinLatitude.setPlaceholder("Minimaler Breitengrad eingeben (47.2701 bis
9      55.0581)");
10 tfMaxLatitude.setPlaceholder("Maximaler Breitengrad eingeben (47.2701 bis
11      55.0581)");
12 tfMinLongitude.setPlaceholder("Minimaler Längengrad eingeben (5.8663 bis
13      15.0419)");
14 tfMaxLongitude.setPlaceholder("Maximaler Längengrad eingeben (5.8663 bis 15.0419)");
15 tfMinLatitude.setValue("50.323");
16 tfMaxLatitude.setValue("52.531");
17 tfMinLongitude.setValue("5.866");
18 tfMaxLongitude.setValue("9.462");
19
20
21
22
23
24 if (ddProvider.getValue() != null) {
25     ddProvider.setValue(null);
26 }
27
28 List<StationDTO> filteredStations = mergedStations.stream()
29     .filter(station -> station.getLatitude() >= minLat && station.getLatitude() <=
30             maxLat
31             && station.getLongitude() >= minLon && station.getLongitude() <= maxLon)
32     .collect(Collectors.toList());
33
34 if (filteredStations.isEmpty()) {
35     Utils.showHinweisBox("Keine Stationen innerhalb der Bounding Box gefunden.");
36 } else {
37     dataProviderStations = stationGrid.setItems(filteredStations);
38     initOpenLayersMap();
39     updateHeaderRows();
40
41     countResult.setText(dataProviderStations.getItemCount() + " Ergebnisse gefunden
42         zu " + tfSearch.getValue());
43 });
44 }
```

```
42 formLayout.setWidth("65%");  
43 formLayout.setResponsiveSteps(new FormLayout.ResponsiveStep("0", 1),  
44 new FormLayout.ResponsiveStep("500px", 2));
```

Dieser Code-Abschnitt erstellt ein Formular zur Filterung von Stationen anhand von Koordinaten (Breitengrad und Längengrad).

Formularkomponenten:

- Vier **Textfelder**: Eingabe von minimalen und maximalen Breiten- und Längengraden.
- Button **Filtern**: Startet den Filterprozess.

Filterlogik beim Button-Klick:

- holt die eingegebenen Koordinaten und filtert die mergedStations-Liste.
- aktualisiert **dataProviderStations** mit den gefilterten Stationen.
- aktualisiert die Karte mit **initOpenLayersMap()** und die Header-Row.

```
1 private Map<String, List<? extends StationDTO>> providerList = new HashMap<>();  
2 ddProvider.setLabel("Servicename(n) auswählen");  
3 ddProvider.setItemLabelGenerator(String::toString);  
4 ddProvider.setWidth("20%");  
5  
6 ddProvider.addValueChangeListener(event -> {  
7     stationGrid.getHeaderRows().clear();  
8     if (event.getValue() != null) {  
9         List<? extends StationDTO> selectedStations =  
10            providerList.get(event.getValue());  
11         if (selectedStations != null) {  
12             tfMaxLatitude.setValue(null && tfMinLongitude.getValue() != null) {  
13                 GridListDataView<StationDTO> dataView = stationGrid.getListDataView();  
14                 if (event.getValue().equals("OpenGeoData-NRW")) {  
15                     dataView.setFilter(st -> selectedStations.contains(st));  
16                     providerColumn.setHeader("Gebiet");  
17                 } else if (event.getValue().equals("Pegelonline-Dienst")) {  
18                     dataView.setFilter(st -> selectedStations.contains(st));  
19                     providerColumn.setHeader("Provider");  
20                 } else {  
21                     dataView.setFilter(st -> st.getProvider().equals(event.getValue()));  
22                     providerColumn.setHeader("Provider");  
23                 }  
24             dataProviderStations = dataView;
```

```

24     } else {
25         if (event.getValue().equals("OpenGeoData-NRW")) {
26             providerColumn.setHeader("Gebiet");
27         } else {
28             providerColumn.setHeader("Provider");
29         }
30         dataProviderStations = stationGrid.setItems(selectedStations.stream()
31             .sorted(Comparator.comparing(StationDTO::getName))
32             .collect(Collectors.toList()));
33     }
34     updateHeaderRows();
35 } else {
36     providerColumn.setHeader("Provider/Gebiet");
37     dataProviderStations = stationGrid.setItems(mergedStations);
38     updateHeaderRows();
39 }
40 stationGrid.getDataProvider().refreshAll();
41 initOpenLayersMap();
42 countResult.setText(dataProviderStations.getItemCount() + " Ergebnisse zu " +
43     tfSearch.getValue()
44     + (event.getValue() != null ? " von " + event.getValue() : ""));
});
```

Dieser Code-Abschnitt implementiert die Filterlogik für den Dropdown-Provider.
Filterlogik bei Wertänderung:

- Überprüft, ob ein Dienstanbieter (`event.getValue()`) ausgewählt ist:
 - falls Koordinatenfilter aktiv: verwendet **GridListDataView**, um die gefilterten Stationen weiter zu filtern.
 - falls kein Koordinatenfilter: sortiert und setzt die Stationen basierend auf dem Dienstanbieter und aktualisiert die Header-Überschrift.
- Falls kein Anbieter ausgewählt ist: setzt die ursprüngliche List als Datensatz und setzt die Standard-Header-Überschrift.
- Aktualisiert die Karte mit **initOpenLayersMap()** und die Header-Zeilen.

GridListDataView ist eine Schnittstelle in Vaadin, die es ermöglicht, die Daten eines Grid-Components zu verwalten und zu manipulieren.

Kapitel 5. Implementierung

Wenn **OpenGeoData-NRW** als Eintrag aus dem Dropdown ausgewählt wird, ergibt sich folgende Abbildung:

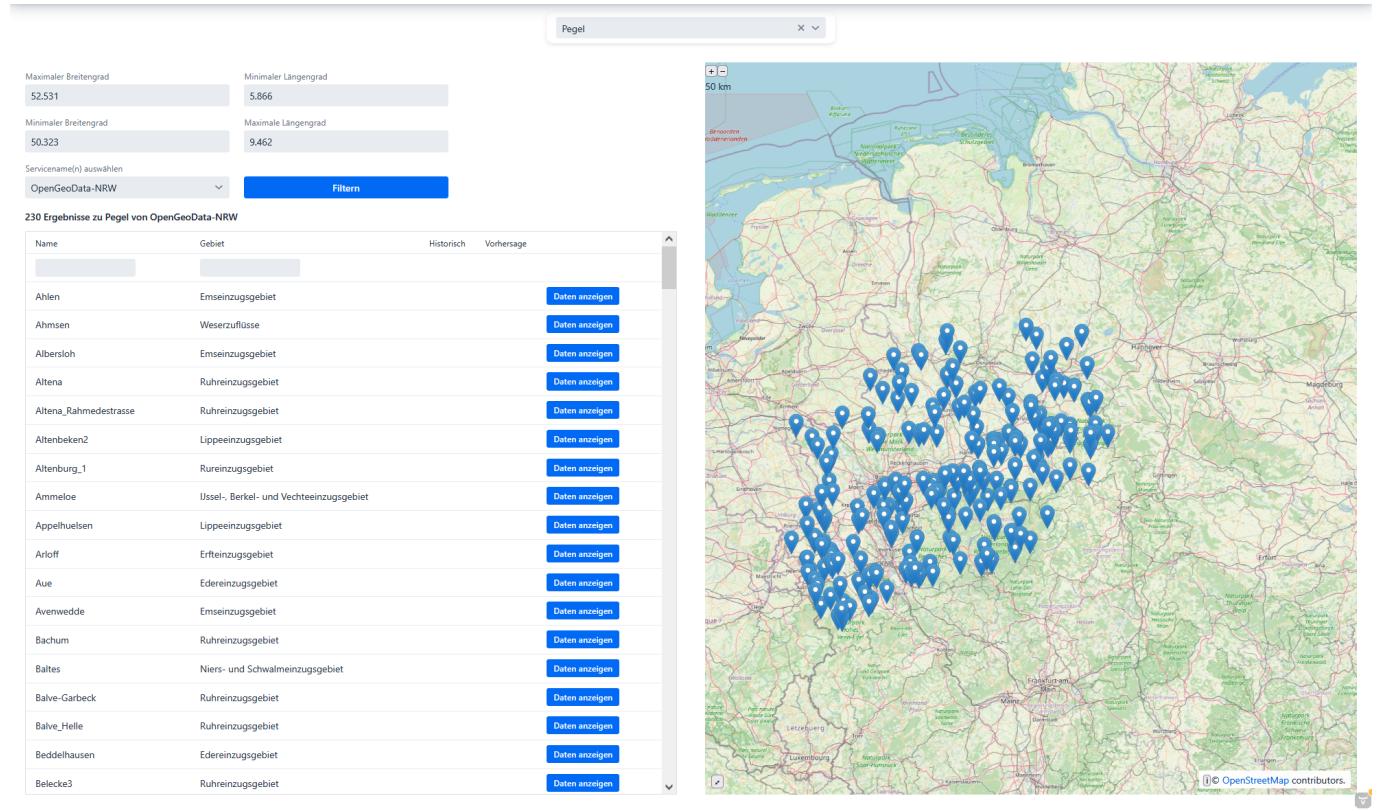


Abbildung 5.3: Pegel-OpenGeoDataNrw

6 Fazit und Ausblick

6.1 Bewertung

Bei der Bewertung zwei grundlegende Aspekte näher betrachtet und kritisch analysiert: die technische Implementierung sowie das Architekturdesign. Der Fokus liegt dabei insbesondere auf der Frage, inwiefern die ursprüngliche formulierten Anforderungen im Verlauf der Entwicklung erfolgreich umgesetzt werden konnten.

Im Rahmen der Arbeit ergeben sich zwei hauptsächlich folgende Forschungsfragen:

- Lässt sich ein generisches Mapping umsetzen, um die heterogenen Datenstrukturen in eine einheitliche Datenstruktur zu überführen.
- Kann eine einheitliche Schnittstelle implementiert werden, die es einer Client-Applikation ermöglicht, in diesem Fall der AhuManager, gezielt einzelnen Datensätze mit dem gewünschten Zielformat zu abonnieren?

6.1.1 Bewertung von Implementierung

Zu der ersten Frage, konnte ein generisches Mapping erfolgreich implementiert werden, um heterogene Datenstrukturen in ein einheitliches Format zu überführen. Dies wurde exemplarisch anhand von drei unterschiedlichen Datenquellen – dem **Deutschen Wetterdienst**, **OpenGeoData.NRW** und **PegelOnline** – umgesetzt. Für diese Quellen funktioniert das Mapping zuverlässig. Allerdings ist die aktuelle Lösung nicht vollständig generisch im Hinblick auf beliebige, unbekannte Datenstrukturen. Für weitere Datenquellen mit stark abweichenden Strukturen sind gegebenenfalls spezifische Anpassungen notwendig, wodurch die generische Anwendbarkeit in solchen Fällen eingeschränkt ist.

Zu der zweiten Frage, konnte eine einheitliche Schnittstelle im Rahmen dieser prototypischen Entwicklung realisiert werden. Diese REST-Schnittstelle ermöglicht es einer Client-Applikation – wie in diesem Fall dem **AhuManager** – gezielt einzelne Datensätze im gewünschten Zielformat zu abonnieren. Derzeit ist die Schnittstelle jedoch nur lokal (localhost) verfügbar, weshalb sie für den produktiven Einsatz noch entsprechend angepasst werden muss, um einen stabilen Datenaustausch zwischen Service und Client auch über Netzwerkgrenzen hinweg zu ermöglichen.

6.1.2 Bewertung von Architekturdesign und Technologien

Um eine Architektur einer Software bewerten zu können, werden unter anderem die **SOLID-Prinzipien** herangezogen. Diese definieren, wie Funktionen und Datenstrukturen in Klassen organisiert sowie wie Abhängigkeiten zwischen Klassen gestaltet sein sollen. Ziel ist die Entwicklung wartbarer, verständlicher und wiederverwendbarer Softwarekomponenten, die als stabile Grundlage für vielfältige Softwaresysteme dienen.

Die fünf SOLID-Prinzipien im Überblick:

- **S** - Single-Responsibility-Prinzip(SRP): Eine Klasse sollte genau eine Verantwortlichkeit besitzen.
- **O** - Open-Closed-Prinzip(OCP): Softwaremodule sollen für Erweiterungen offen, aber für Modifikationen geschlossen sein.
- **L** - Liskov'sche Substitutions-Prinzip(LSP): Objekte von Subklassen sollen sich wie Objekte ihrer Superklasse verhalten lassen.
- **I** - Interface-Segregation-Prinzip(ISP): Schnittstellen sollen spezifisch auf konkrete Anforderungen zugeschnitten sein.
- **D** - Dependency-Inversion-Prinzip(DIP): Abhängigkeiten sollen sich auf Abstraktionen, nicht auf konkrete Implementierung beziehen.

Im vorliegenden Projekt wurden mehrere dieser Prinzipien umgesetzt.

- **Single-Responsibility-Prinzip**: Wie bereits in Abbildung 4.2 dargestellt, wurde dieses Prinzip konsequent umgesetzt. Jede Komponente bzw. jeder Service ist in einem eigenen Package gekapselt und übernimmt eine klar abgegrenzte Aufgabe.
- **Liskov Substitution Principle**: Auch dieses Prinzip wurde berücksichtigt. So stellt z.B. die Klasse „ **BaseEntity**“ eine abstrakte Oberklasse dar, deren Subklassen ihre zugesicherten Eigenschaften einhalten. Die Anwendung dieses Prinzips wurde im Detail in Abschnitt 5.1 erläutert.
- **Dependency-Inversion-Prinzip**: Durch den Einsatz des Spring Frameworks wird dieses Prinzip weitgehend automatisch unterstützt. Insbesondere durch die Verwendung der Annotation **@Autowire** erfolgen Abhängigkeiten auf Basis von Interfaces bzw. Abstraktionen und nicht auf konkrete Implementierungen.

Für die Umsetzung der spezifizierten Anforderungen wurden bewährte Technologien eingesetzt, die eine effiziente Entwicklung und eine benutzerfreundliche Darstellung der Ergebnisse ermöglichen.

Im Backend kam das Spring Framework zum Einsatz, das durch seine modulare Struktur, umfangreiche Konfigurationsmöglichkeiten und integrierte Unterstützung für Dependency Injection eine schnelle und strukturierte Implementierung der Geschäftslogik erlaubt.

Für die Gestaltung der Benutzeroberfläche wurde Vaadin verwendet. Diese Java-basierte Web-UI-Technologie bietet eine Vielzahl an vorkonfigurierten Komponenten, die sich besonders gut zur Darstellung großer Datenmengen eignen. Durch die hohe Abstraktionsebene und serverseitige Verarbeitung lassen sich komplexe Benutzeroberflächen mit minimalem Aufwand realisieren. Insbesondere die Möglichkeit, Messdaten übersichtlich und interaktiv anzuzeigen, stellt einen wesentlichen Vorteil von Vaadin im vorliegenden Projekt dar.

6.2 Ausblick

Die zentrale Fragestellung zu Beginn dieser Arbeit lautete, ob die prototypische Entwicklung zur Messdatenerhebung und -bereitstellung einen konkreten Mehrwert für den **Ahu-Manager** liefern kann. Diese Frage lässt sich nicht abschließend beantworten, bietet jedoch Diskussionspotenzial. Der entwickelte Prototyp bildet eine solide Grundlage, auf der weiter aufgebaut werden kann.

Insbesondere in folgenden Punkten besteht Weiterentwicklungspotenzial:

- **Ausbau der Schnittstellen:** Die Anbindung eines stabilen und skalierbaren Abonnement-Services könnte die Datenbereitstellung effizienter und flexibler gestalten.
- **Integration weiterer Datenquellen:** Eine Erweiterung auf zusätzliche externe Systeme würde Mehrwert bringen.
- **Verbesserung der Benutzeroberfläche:** Die grafische Darstellung und Interaktivität der Oberfläche lässt sich im Hinblick auf Usability und Performance weiter optimieren.

Ein spannender Aspekt für die zukünftige Entwicklung wäre die Frage, ob sich eine **generalisierte Datenstruktur** realisieren lässt, in der alle abonnierten Daten einheitlich in einer Datenbank abgelegt werden können – beispielsweise angelehnt an das **SensorThings API-Modell**. Eine solche Standardisierung könnte die Interoperabilität zwischen verschiedenen Systemen wesentlich verbessern.

Literatur

Deinhard, F. (2024). *Was ist Maven?* Verfügbar 1. September 2024 unter <https://www.itschulungen.com/wir-ueber-uns/wissensblog/was-ist-maven.html>

Document. (2021). *Overview*. Verfügbar 3. Februar 2021 unter <https://vaadin.com/docs/v8/framework/architecture/architecture-overview>