Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Konzeption und prototypische Entwicklung einer Webanwendung zur Aggregation und Analyse wissenschaftlicher Daten des Argo-Projektes

Sebastian Schmid
S0543196

Prof. Dr. Christin Schmidt
Prof. Dr.-Ing. Hendrik Gärtner

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik
an der Hochschule für Technik und Wirtschaft Berlin

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum Unterschrift

Danksagung

An dieser Stelle möchte ich meiner Betreuerin Frau Prof. Dr. Christin Schmidt danken. Diese unterstützte mich stets tatkräftig. Auch Herrn Prof. Dr.-Ing. Gärtner möchte ich dafür danken, dass er sich dazu bereit erklärt hat, diese Arbeit zu betreuen.
Außerdem möchte ich auch meinen Eltern danken. Auch wenn diese heute nicht mehr unter uns sind, so waren sie es doch, die den Keim gelegt haben, der mich dazu befähigte den Weg bis zu dieser Stelle zu gehen.
Zu danken habe ich auch Robert M. Pirsig. Sein Buch „Zen und die Kunst ein Motorrad zu warten“ begleitete mich bei der Entwicklung dieser Arbeit und inspirierte mich immer wieder aufs Neue. Ich könnte mir kein besseres Buch als Begleitung für eine schriftliche Ausarbeitung vorstellen.
Zuletzt und doch ganz besonders richtet sich mein Dank an Denise Peth. Diese begleitete mich auf dem Weg dieser Ausarbeitung und stand mir stets mit Rat und Tat zur Seite.

Inhalt

Abbildungsverzeichnis

Quellcodeverzeichnis

Tabellenverzeichnis

1 Einleitung

Die im Jahre 1965 von Gordon Moore vorhergesagte Gesätzmäßigkeit, die Komplexität, und damit die Speicherdichte integrierter Schaltkreise, würde sich regelmäßig verdoppeln (vgl. [Moo98]), hat sich seitdem bis zum heutigen Tag bewahrheitet. Damit sieht sich die Menschheit über 50 Jahre später in einer einmaligen Lage. Zum einen verfügen wir über eine noch nie dagewesene Ansammlung an Informationen, zum anderen wurden die Speichermedien durch die wachsende Komplexität immer flüchtiger und schwieriger in der technischen Handhabung. Werden kommende Zivilisationen in der Lage sein, diesen Pool an Informationen für sich zu nutzen oder sollten wir vielmehr annehmen, dass wir eine einmalige Chance haben, die wir nicht vergeuden sollten?

In diesem Kontext scheint es als folgerichtig, dass Bewegungen wie Freie Software, Creative-Commons und Open-Access die Hürden für den Zugang zu den Informationen weiter verringern. Viele der Informationen stehen heute frei zur Verfügung und können genutzt werden und doch werden, von einem gefühlt immer größeren Anteil unseres Kulturkreises, Prinzipien zur Bewertung von Quellen abgelehnt. In einer Zeit in der es einfacher denn je ist, Fakten zu überprüfen, werden wissenschaftlich bewiesenen Aussagen wie dem Klimawandel nicht geglaubt. Es scheint als wären viele Menschen der Flut an Informationen überdrüssig, als wende sich ein großer Teil überfordert davon ab. Die Frage ist, was kann dazu beitragen dieses Potential mehr zu nutzen? Über welche Mittel verfügt die Informatik, Daten in einen Kontext einzubetten? Wie muss der Kontext gestaltet sein, über den Menschen an wissenschaftliche Arbeit herangeführt werden können? Welche Daten sind geeignet, um die Brisanz und das Potential unserer Zeit einem breiteren Publikum zuzuführen? Mit diesen Fragestellungen motiviert sich die hier vorliegende Arbeit.
Im Rahmen dieser Abschlussarbeit wurde, über die Aggregation wissenschaftlicher Daten, ein exploratives Werkzeug geschaffen, um diese intuitiv erfahrbar zu gestalten. Dies geschieht mit dem Ziel, dass hier eine Identifikation mit den Messwerten und des wissenschaftlichen Prozesses zu erreicht werden kann. Als Datengrundlage wurden Daten des Argo-Programms verwendet. Unter dem Dach dieses Forschungsprojektes werden seit dem Beginn dieses Jahrtausends das Wasser der Weltmeere nach den Parametern Temperatur, Salzgehalt und Leitfähigkeit untersucht. Diese Daten stehen unter einer freien Lizenz zur Verfügung und können in eigene Projekte eingebunden werden. Die Messdaten dienen Wissenschaftlern, die Auswirkungen des globalen Klimawandels zu untersuchen. Die Herausforderung besteht darin, die hochkomplexen Messdaten in einer Form aufzubereiten, so dass diese einfach verstanden werden können.

Ein solches Werkzeug könnte zum Beispiel in Schulklassen eingesetzt werden. Die Ler-

nenden könnten, innerhalb einer Kontextvorgabe durch den Lehrenden, die Auswirkungen des Klimawandels auf die Weltmeere untersuchen. Hier könnte auch das Interesse und ein eventueller Berufswunsch im Wissenschaftssektor geweckt werden. Die Applikation richtet sich aber nicht explizit an Heranwachsende. Erwachsene Personen könnten sich hier, eigenes Interesse vorausgesetzt, weiterbilden. Die Applikation kann hier Relevanz der Forschung und die in diesem Programm erhobenen Messwerte erfahrbar machen.

Der schriftliche Teil dieser Arbeit handelt von der Konzeption und Entwicklung eines Prototypen mit der oben genannten Problemstellung. Zu Beginn wird das Argo-Programm näher vorgestellt. Es wird auf den Prozess der Datenerhebung und Aufbereitung eingegangen. Im Anschluss daran werden die Anforderungen der Software ausgearbeitet, um im Folgenden darauf gangbare Lösungsansätze und die geeignetsten Werkzeuge ermittelt. Anschließend wird das Softwaresystem entworfen. Es werden die geplante Architektur und die wichtigsten Geschäftsprozesse ausgearbeitet und beschrieben. Über das Aufzeigen von gangbaren Alternativen werden die passendsten Werkzeuge und Entwicklungsmuster für diese Problemstellung ermittelt. Nach der Planung folgt eine Beschreibung der vorgenommenen Implementierung. Auch hier werden Implementierungen iterativ verfeinert, so das die passendste Lösung gefunden werden kann.
Um die Qualität der Software beschreiben zu können, folgt im darauf folgenden Kapitel eine Beschreibung der verwendeten Testverfahren. Hier werden Metriken ausgearbeitet, welche erlauben, funktionale als auch nicht-funktionale Anforderungen überprüfbar messen zu können.6
Im Abschluss findet eine Präsentation des entwickelten Prototypen statt. Dabei wird ein möglichst kritischer Blick auf Problemstellung und Lösungen geworfen. Abschließend wird ein Ausblick auf Alternativen und mögliche Weiterentwicklungen der Prototypen gegeben.

1.1 Existierende Lösungen

Das Joint Technical Commission for Oceanography and Marine Meteorology (JCOMM) bietet mit `jcommops.org` eine Grundlage für die wissenschaftliche Arbeit unter anderem mit den von Argo gesammelten Daten. Neben der Darstellung von Karten erfährt der Nutzer hier, von Sensordaten über den Bautyp der jeweiligen Boje alles, was die Bojen zu erzählen haben. Daneben werden über diese Plattform auch redaktionelle Reports veröffentlicht, um der Leserschaft ein Bild der aktuellen Lage unserer Weltmeere zu vermitteln. Zwar bietet die Plattform durch ihre kartenbasierte explorative Darstellung ein ähnliches Angebot, wie es in der hier zu erstellenden Applikation geben soll. Die Fülle der Parameter erfordert vom Benutzer aber die Bereitschaft, sich vertieft in die Matherie einzuarbeiten. Damit richtet sich das Angebot des Global Ocean Observing Systems an Wissenschaft-

ler und Journalisten. Diesen dient sie als eine hervorragende Datengrundlage für deren Veröffentlichungen. Ein Benutzer aus der für diese Arbeit definierten Zielgruppe wird von einem derartigen Angebot wohl eher abgeschreckt sein, bevor er dessen Vorteile für sich erarbeitet.

Das „Data Selection and Visualization Tool“ der Coriolis GDAC (siehe [Arg17b]) ermöglicht den Download von, nach verschiedenen Filterkriterien ausgewählten, Datensätzen. Über einen Betrachter können Werte einer spezifischen Treibboje angesehen werden.
Die primäre Aufgabenstellung dieser Plattform ist das Extrahieren der Daten, um diese in einer wissenschaftlichen Publikation verwenden zu können. Neben der akademischen Verwendung müsste ein Benutzer aber auch hier viel Neugierde und Zeit für die Einarbeitung in das Thema mitbringen, um aus den angebotenen Daten einen Mehrwert für sich zu generieren.

1.2 Alleinstellungsmerkmal & Abgrenzung

Das Ziel dieser Arbeit ist die Darstellung von wissenschaftlichen Daten aus dem Argo-Programm. Im Gegensatz zu den bereits vorhandenen Lösungen sollen die Daten aber nicht zur wissenschaftlichen Verwendung aufbereitet werden. Vielmehr sollen die hochkomplexen Daten auf ein einfach erfahrbares Maß herunter gebrochen werden. Zum Zeitpunkt dieser Arbeit existiert noch kein exploratives Tool für Daten des Argo-Programms für die hier definierte Zielgruppe.

2 Grundlagen

2.1 Argo-Programm

Zum Ende des vergangenen Jahrtausends verdichteten sich die Hinweise auf einen globalen und durch Menschen verursachten Klimawandel. Um dessen Auswirkungen auf die Weltmeere studieren zu können, wurde unter dem Dach des Global Ocean Observing System das Argo Programm gegründet. Dieses untersucht, unterstützt durch das Satellitensystem Iason, die Wassersäule der oberen 2000m auf deren chemischen Eigenschaften. Dabei werden in ständigen Intervallen Salzgehalt, Druck, Temperatur und Leitfähigkeit gemessen. Die ermittelten Daten werden veröffentlicht, dass diese durch Wissenschaftler ausgewertet werden können.

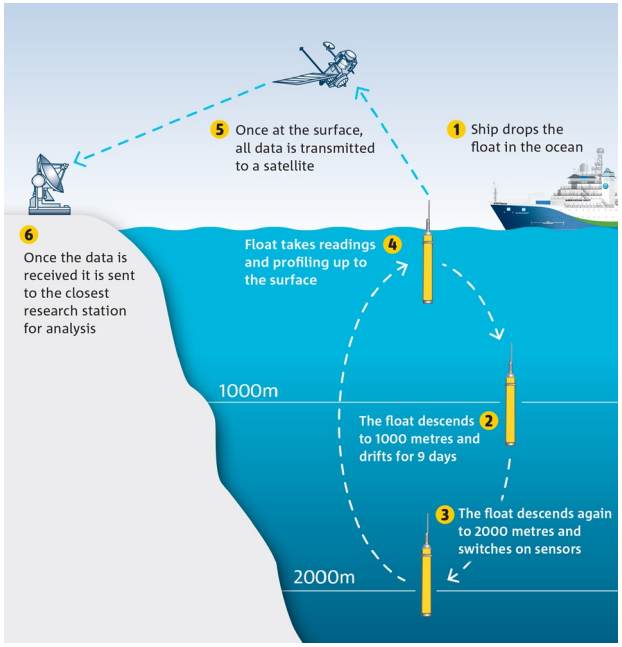

Abbildung 1: Argos Messzyklus
Bildquelle: `https://www.csiro.au`

Die Argo Treibbojen werden mit Schiffen an spezifischen Punkten ausgesetzt. Ein Messzyklus beträgt 10 Tage. Die Boje taucht am Anfang des Zyklus auf 1000 Meter Tiefe herab. In dieser Tiefe verbringt die Sonde die nächsten 9 Tage. Anschließend sinkt sie auf die maximale Tiefe von 2000 Metern herab, um anschließend wieder zur Oberfläche aufzusteigen. An der Oberfläche sendet die Boje innerhalb von 6 bis 12 Stunden die Daten über den Sattelitenarray Iason an die Bodenstationen. Der oben genannte Messzyklus kann in Abbildung 1 nachvollzogen werden.
Im Anschluss an die Erhebung werden die Daten auf deren Plausibilität und Qualität überprüft (vgl. [SG] S. 3). Nach diesem Prozess werden die aufbereiteten Daten über die

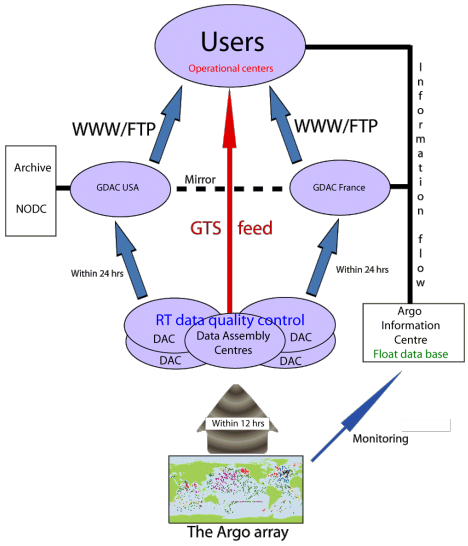

Abbildung 2: Qualitätszyklus der Argo-Daten
Bildquelle: `http://www.argo.ucsd.edu`

Globalen Data Assembly Center in monatlichen Releasezyklen veröffentlicht. (vgl. [Arg]). Der Werdegang der Daten ist in Abbildung 2 zu sehen.

2.2 Datengrundlage

Alle am Argo teilnehmenden Organisationen verpflichten sich auf eine gemeinsame Datenpolitik. So gibt es keine Herrschaft über die erhobenen Daten. Vielmehr stehen diese ab dem Zeitpunkt der Veröffentlichung transparent der Öffentlichkeit zur Verfügung.
Nach der Übermittlung, werden die Messdaten einer Qualitätskontrolle unterzogen. Sie werden auf Plausibilität und Abweichungen überprüft. Nach dieser Qualitätskontrolle werden die Daten nun über die Global Data Assembly Centers (GDAC) in Frankreich und den USA released. Diese können über HTTP und FTP abgerufen werden.
Ein mal Im Monat werden die Daten als Snapshots, den sogenannten DOI (digital object identifier) zusammengefasst. Während und vor der Qualitätskontrolle liegen die Daten in den Formaten TESAC und BUFR vor. Die von den DGAC veröffentlichten Daten liegen im Format netCDF vor. Diese sind unter der Lizenz Attribution 4.0 International (CC BY 4.0) veröffentlicht und dürfen unter der Nennung der Lizenz frei verwendet und dabei auch verändert werden.
[Arg17a]
In Listing 1 ist die Ordnerstruktur der netCDF Dateien zu erkennen. Über den Ordnernamen aoml ist die Herkunft des DOI zu erkennen. In diesem Fall wurden die Dateien von einem Server der „Atlantic Oceanografic & Meterolocical Laboratory“ erstellt. Hier befindet sich für jede Messboje ein Unterordner. Die Dateien meta, prof, Rtraj und tech

sind eine Quelle für Metainformationen. Die Messprofile einer Boje finden sich im Ordner `profiles`. Hier wird für jeden Messzyklus eine Datei angelegt. Dieser startet bei 1 und inkrementiert über jeden Messzyklus um 1.

```
./aoml/1900200/
- 1900200_meta.nc
- 1900200_prof.nc
- 1900200_Rtraj.nc
- 1900200_tech.nc
- profiles
    - D1900200_001.nc
    ...
    - D1900200_215.nc
    - D1900200_216.nc
./aoml/1900201/
...
```

Listing 1: Die Verzeichnisstruktur der vom aoml bereitgestellten Daten

2.3 Objekt-Relationale Unverträglichkeit

In einem Softwareparadigma manifestiert sich ein bestimmtes Konzept in der Modellierung der Welt. Durch diese konzeptionelle Betrachtungsweise beeinflusst ein Paradigma den Erstellungsprozess sowie die Ergebnisse des Softwaredesigns und zwingt diesem seine Grenzen auf. Probleme bei der Kombination verschiedener Paradigmen werden dabei als Unverträglichkeit (impedance Mismatch) bezeichnet. Objektorientierte Programmierung und eine relationale Abfrage von Datensätzen sind weit verbreitete Paradigmen in der Softwareentwicklung. Damit ist die objekt-relationale Unverträglichkeit eine häufig auftretende Herausforderung, (vgl. [IBNW09] S. 36-38). Hier genannt sind unter anderem folgende Betrachtungsweisen als Ursachen für die objektrelationale Unverträglichkeit:

Strukturelle Unterschiede Die objektorientierte Programmierung erlaubt das Definieren, beliebig komplexer Strukturen aus Methoden und Klassen. Durch Vererbungsstrukturen ist es möglich, Objekte zu spezialisieren und Konzepte für die Erstellung der Klassen zu generalisieren. Eine relationale Algebra wird durch Tupel, Mengen und Wahrheitswerte definiert. Inhärent wiederholbare Strukturen oder Hierarchien können mit diesen Mitteln nicht umgesetzt werden.

Datenkapselung In der objektorientierten Programmierung können die intrinsischen Attribute eines Objektes verborgen werden. Dieses Konzept ist als Kapselung bekannt und erlaubt es den Zugriff auf die gekapselten Strukturen einzuschränken. In einer relationalen Algebra ist eine derartige Abstraktion über die Daten nicht vorgesehen.

Objektidentität Durch die Instanziierung eines Objektes aus einer Klasse erhält dieses eine eindeutige Identität. Somit unterscheiden sich zwei Objekte, auch wenn diese Träger eines identischen Datensatzes sind, durch ihre Repräsentation im Arbeitsspeicher. Relationen werden durch den Primärschlüssel, und damit über ihre Daten, definiert. Zwei eigenständige Relationen mit identischem Datensatz sind damit nicht möglich.

Als Hilfsmittel zur Überwindung der oben genannten Probleme werden objektrealtionale Mapper (ORM bzw. O/R-Mapper) eingesetzt. Durch dieses Mapping wird eine Schnittstelle oder Abstraktionsebene zwischen Programmteilen aus den jeweiligen Sprachparadigmen definiert, um den impedance Mismatch möglichst transparent zu überwinden.

2.4 Geodäsie und Kartographie

Die geographsiche Länge eines Punktes P_l bezeichnet den Winkel zwischen einer vom Nullmeridian durch den Erdmittelpunkt geführten Fläche und einer Meridianebene die durch den Punkt P_l führt. Analog dazu ist die geographische Breite eines Punktes P_b der Winkel zwischen der Äquatorschnittfläche und der Flächennormalen an Punkt P_b (vgl. [WS11] S. 18).
Die Kartographie hat das zentrale Problem, dass es sich bei Karten immer um eine zweidimensionale Projektion einer Sphäre handelt. Mit einem Netzentwurf kann eine Karte nur zwischen den Polen Winkel-, Längen- oder Flächentreue zu jeweils einer Seite hin optimiert werden. Frei von Verzerrungen ist nur ein Globus. Längentreue Darstellungen können mit azimutalen Netzentwürfen realisiert werden. Dabei wird eine Tangentialebene an den Globus gelegt. Auf diese werden werden die Netzlinien um ein Zentrum herum projiziert. Winkeltreie Projektion können beispielweise mit gnomischen Abbildungen in einem polaren Koordinatensystem erreicht werden. Konische oder zylindrische Abbildungen projizieren das Kartennetz auf einen Kegelmantel bevor die Ebene abgewickelt wird. Durch diese Projektionsform können flächentreue Abbildungen erreicht werden.
(vgl. [WS11] S. 503-506)
Das „World Geodetic System 1984 (WGS84)“ ist ein geozentrisches, vom Erdschwerpunkt abgeleitetes Koordinatensystem. Dieses wird unter anderem von GPS zur Zuordnung von Positionen auf der Erde verwendet. Eine schematische Darstellung ist in Abbildung 3 zu sehen. Die Z-Achse führt durch den Erdschwerpunkt entlang der Rotationsachse durch den Nordpol, die X-Achse von Zentrum der Erde durch einen Nullmeridian (Greenwich) und die Y-Achse wird als Orthogonale zur X-Achse ebenso durch den Erdschwerpunkt geführt (vgl. [WS11] S. 13-14).
Die Gauß-Krüger Projektion erlaubt die Darstellung des Globus auf einer Ebene. Um die

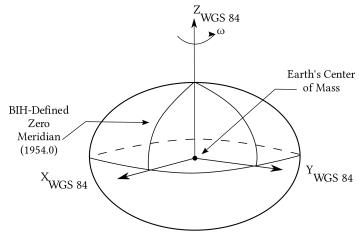

Abbildung 3: Schematische Darstellung des Koordinatensystems WGS84
Bildquelle: https://commons.wikimedia.org

Krümmung des Globus auszugleichen wird die Erde in $3°$ breite Meridianstreifen eingeteilt. Als MIttelpunkte dienen Dabei die Meridiane $[3°, 6°, 9°, \ldots]$. Von den Mittelmeridianen aus wird eine Projetion an anliegende Zylinder vorgenommen. Auf diese Weise entstehen zu den Rändern der meridianstreifen Längen- und Flächenverzerrungen, die Darstellung ist aber Winkeltreu. Im UTM-System wird ein analoges Verfahren angewendet. Die Meridianstreifen haben hier aber eine Breite von $6°$. Anstatt dem Mittelmeridian werden hier zwei Schnittkurven mit einem Abstand von 180km als Längentreues Element für die Abbilder der Y-Achse verwendet. Der Bereich zwischen diesen Schnittkurven wird gestaucht. (vgl. [WS11] S. 23-25)

2.5 Network Common Data Format

Das Network Common Data Format (netCDF) dient zum Austausch von wissenschaftlichen Daten. Es ist eine Weiterentwicklung des von der NASA entwickelten Common Data Formats (CDF). Das Datenformat zeichnet sich dadurch aus, dass es selbstbeschreibend ist. Die Dokumentation ist Teil des Datensatzes und wird so immer mitgeführt. Dies soll die Portabilität des Datensatzes verbessern. (vgl. [AGS+])
Neben einigen anderen Sprachen, wurde auch eine Bibliothek für Python entwickelt (vgl.[net]) Mit dieser ist es möglich, die Binärdaten zu öffnen und zu numpy-Arrays zu extrahieren. In Listing 2 ist die Verwendung exemplarisch an der Extraktion und der Umrechnung des Erstelldatums aufgezeigt und im Folgenden beschrieben.

```
1  from netCDF4 import Dataset
2  import datetime
3
4  dataset = Dataset('./4900442/profiles/D4900442_042.nc')
5  print(dataset.variables['JULD'])
6
7  # <class 'netCDF4._netCDF4.Variable'>
8  # float64 JULD(N_PROF)
9  #     long_name: Julian day (UTC) of the station relative to
       ↪ REFERENCE_DATE_TIME
10 #     units: days since 1950-01-01 00:00:00 UTC
11 #     conventions: Relative julian days with decimal part (as parts of
```

```
    ↪ day)
#     _FillValue: 999999.0
# unlimited dimensions:
# current shape = (1,)
# filling off

julian_date = dataset.variables['JULD'][::][0]
dataset.close()

print(julian_date)
# 20062.5483218

juld_zero = datetime.datetime.strptime( '1950-01-01 00:00:00 UTC',
                                        '%Y-%m-%d %H:%M:%S UTC')
date_creation = juld_zero + datetime.timedelta(days=int(julian_date))
print(date_creation)
# datetime.datetime(2004, 12, 5, 0, 0)
```

Listing 2: Extraktion und Berechnung des Erstellungsdatums eines NetCDF-Datensatzes

Um einen Datensatz zu öffnen, bildet man eine Instanz der Klasse `netCDF4.Dataset`. Die Auswahl des Profils gelingt durch die Pfadangabe als Parameter bei der Instanzierung.
Über das Attribut `dataset.variables` werden die Datensätze als `OrderedDict` gehalten. Bei der Extraktion eines Parameters erhält man zunächst die Dokumentation des jeweiligen Datensatzes. In diesem Fall handelt es sich um den Zeitpunkt der Satteliten-übertragung des betreffenden Messprofils. Aus der Dokumentation lassen sich für die Weiterverarbeitung wichtige Parameter entnehmen. So ist der Datensatz in Form des C-Datentyps `float64` codiert. Beim Datumsformat handelt es sich um *Julian day* ab dem Zeitpunkt *01. Januar 1950*.
Um die Werte eines Datensatzes zu extrahieren, wird über die numpy-slicing Operation `arr[::]` der Datensatz in vektorieller Form extrahiert. Da in diesem Array nur ein skalarer `float64` Wert enthalten ist, kann dieser als nulltes Element vom Array entnommen werden.
Zur Überführung in in das Format des Gregorianischen Kalenders wird ein Datumsobjekt des Referenzdatums benötigt. Durch die Verwendung von `timedelta` werden die Tage aus dem Feld `JULD` addiert. Da ein Messzyklus 10 Tage andauert, kann an dieser Stelle der Datensatz, durch das Überführen zu einem Integer, vereinfacht werden. Die Casting-Operation `int()` rundet in jedem Fall ab.

3 Anforderungsanalyse

3.1 Systembeschreibung

Die zu entwickelnde Anwendung verfolgt einen datengetriebenen Ansatz. Daten aus dem Argo-Programm werden erhoben und eine Auswahl davon in ein Datenformat überführt. Zur Darstellung wird eine Webapplikation verwendet. Über diese werden über Postionsmarker auf einer Weltkarte, die letzten Positionen der Messstationen angezeigt. Über einen Klick auf die Repräsentation einer Messboje werden weitere Daten der Messstation präsentiert.

3.2 Ermittlung der Anforderungen

Ausgehend von einem ersten Prototypen wurden Use-Cases entwickelt. Eine Schematische Darstellung ist in Abbildung 4 zu sehen. Dabei wurden zwei Klassen von Akteuren herausgearbeitet. Darunter finden sich die Benutzenden, also die eigentlichen Anwender der Applikation. Diese erwarten einen reibungslosen Betrieb der Applikation und die Darstellung der Messstationen in der gegebenen Form. Zum anderen existiert die Klasse der Administrierenden. Diese benötigen Werkzeuge, um die Daten zu restrukturieren und in die Datenhaltung einzuspielen. Beide Klassen haben verschiedene Sichten auf die Datenhaltung. Daraus entsteht ein Spannungsverhältnis, welches minimiert werden muss.

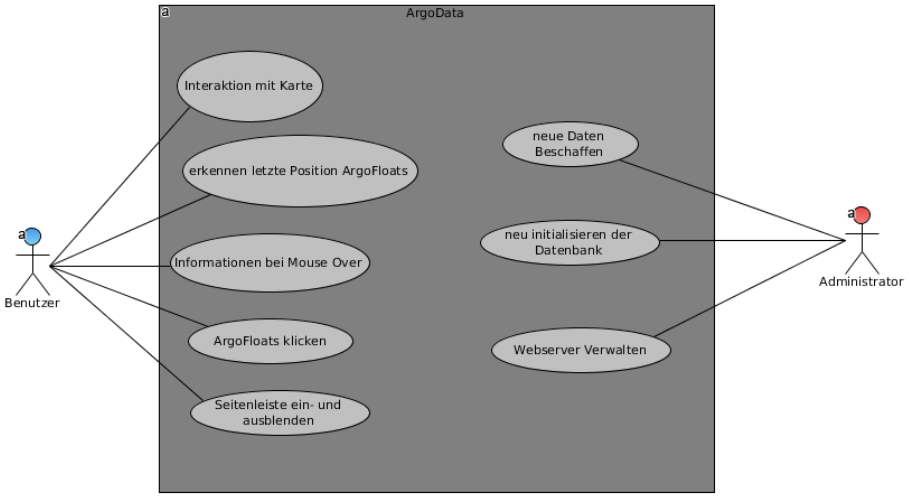

Abbildung 4: Use case Diagramm der Anforderungen

Anhand dieses Modelles wurden Anforderungen ausgearbeitet. Diese wurden in Form eines , in Abbildung 5 ersichtlichen, gerichteten Graphen aufbereitet. Dabei werden die Anforderungen und deren Nachbedingungen über Knoten des Graphen repräsentiert. Finden sich in einer Anforderung Vorbedingungen zu den Nachbedingungen einer anderen

Anforderung, so wird diese Abhängigkeit über eine Kante dargestellt. Diese Darstellung half dabei, Definitionslücken in den Beziehungen der Anforderungskette besser verstehen zu können und formale Lücken schließen zu können.

3.3 Funktionale Anforderungen

3.3.1 Anwendende Perspektive

In der Anwendung wird die Darstellung einer Webapplikation erwartet. Angesteuert durch die zugehörige URL, sollen über eine Kartenapplikation die Grenzen der Ozeane und der umliegenden Kontinente erkennbar werden. Eine genauere Darstellung von Landmarken wie Straßen, Städte oder Gebirge ist für den Aussagewert der Applikation nicht erforderlich. Die Kartendarstellung soll über Interaktionsmöglichkeiten, wie dem Einstellen der Zoomstufe, sowie des ausgewählten Kartenbereichs, verfügen. Über den Kartendienst sind die letzte Position jeder Messboje aus dem Datensatz ersichtlich. Um Streuung und Verdichtung an spezifischen Orten sehen zu können, werden alle Bojen des angezeigten Kartenausschnittes dargestellt. Es erfolgt keine Zusammenfassung oder Ausblendung. Wird die Maus über die visuelle Repräsentation einer Messboje geführt, so sollen grundlegende Daten der Messstationen dargestellt werden, gleichzeitig wird diese Station optisch hervorgehoben. Ein Mausklick auf die schematische Darstellung soll hier gemessene, spezifische Daten anzeigen. Dazu werden in einem separierten Darstellungsfeld der Verlauf der Messdaten, sowie weiterere sinnvolle Werte angezeigt. Zusätzlich soll der zurückgelegte Weg der Messtation ersichtlich sein. Eine Trendanalyse der Daten kann hier ein Maß für die Richtung der Messwerte ermöglichen. Können mehrere Messstationen gleichzeitig ausgewählt werden, so wäre es sinnvoll die akkumulierten Werte darzustellen.

3.3.2 Administrative Perspektive

Es notwendig, dass die Datensätze des Argo Programms ausgelesen und in ein für die Datenbank aufbereitetes Format überführt werden. Neue Daten müssen hierfür aus den Quellen des Argo Programms heruntergeladen werden. Um die Inhalte zu erstellen, wird statt einer journalistischen eine administrative Tätigkeit erwartet. Durch den Datengetriebenen Ansatz der Applikation müssen für das Erstellen der Inhalte keine Webmasken wie Texteingabefelder zur Verfügung gestellt werden. Vielmehr ist es sinnvoll, die hier benötigten Werkzeuge als Skripte zur Verfügung zu stellen, um die administrativen Aufgaben über die Kommandozeile ausführen zu können. Dies würde eine Abbildung der Prozesse bis hin zu einer Vollautomatisierung erlauben und eine örtliche Trennung von Datenaggregation und -darstellung vereinfachen. Für die administrativen Tätigkeiten werden Werkzeuge benötigt, um neue Daten in die Datenbank überführen können.

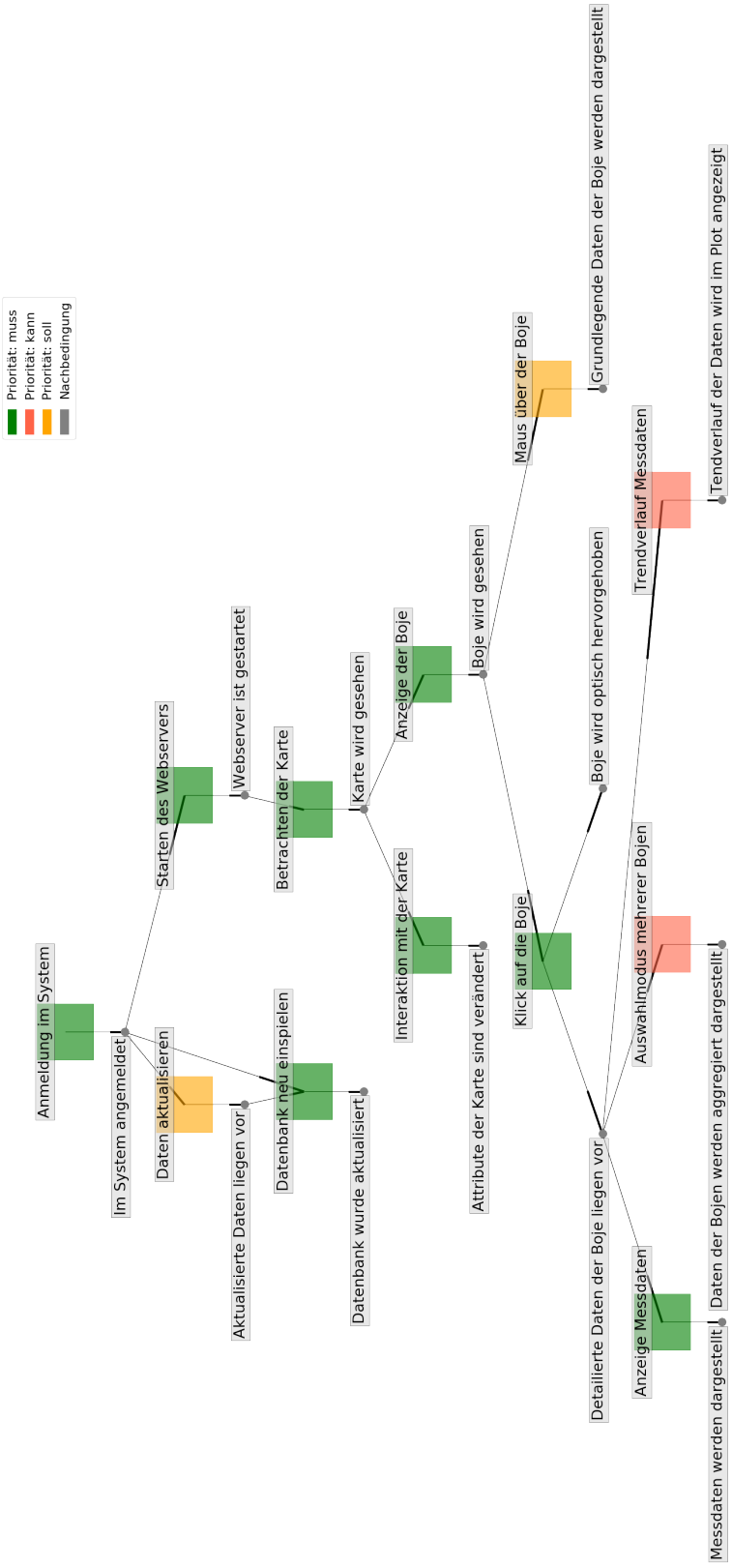

Abbildung 5: Anforderungen und ihre Abhängigkeiten als gerichteter Graph

3.4 Nicht-Funktionale Anforderungen

3.4.1 Anwendende Perspektive

Für die Benutzenden sind Usability und Benutzbarkeit der wichtigste Aspekt. Die Bedienungsmuster müssen klar ersichtlich und intuitiv erfahrbar sein. Die Darstellung der Applikation soll einfach und schlicht gehalten werden und dem gewohnten Erscheinen modernen Web Applikationen entsprechen. Es soll Wert auf die wichtigsten Aspekte des Datenschutzes Wert gelegt werden. Insbesondere im Hinblick auf Datensparsamkeit sind das Einbinden von Trackern, Loginfunktionen und das Speichern von Logfiles zu bewerten. Sicherheitsaspekten ist aus der Anwendersicht nur eine normale Rolle zugewiesen werden. Dabei sollen die aktuell als Standard angesehenen Sicherheitsvorkehrungen getroffen werden.

3.4.2 Administrative Perspektive

Für die Aggregation der Daten ist die Sicherheit ein zentraler Aspekt. Dies umfasst Aspekte von Authentifizierung und Autorisierung der Rolle der Datenaggregation als auch eine Sicherheit um die Integrität der heruntergeladenen Daten.

3.5 Benötigte Daten

Um die Darstellung zu vereinfachen und die Kommunikation mit der Datenbank zu beschleunigen, ist im ersten Schritt eine Auswahl aus den Daten der Argo-Bojen vorzunehmen. Hierbei soll darauf geachtet werden, nur diejenigen Daten zu verwenden, welche für die Erbringung des Dienstes notwendig sind. Aus diesem Grund findet sich Tabelle 1 eine Auswahl aus dem Datenkatalog (vgl. [CKT+15] S. 19 ff.) des Argo-Programms.

Das Feld PLATFORM_NUMBER wird benötigt um Messprofile eindeutig der Plattform zuordnen zu können, während die CYCLE_NUMBER eine Messung innerhalb ihrer Plattform eindeutig zuordenbar macht. Das Datum der Messdatenübertragung kann aus dem Feld JULD ermittelt werden. Dieses erlaubt es, die Messreihe in ihren zeitlichen Kontext zu sehen. Die Anzahl der Sensoren lässt sich aus dem Feld N_PARAM herleiten. Dies wird benötigt, da einige Messstationen nicht alle Sensoren eingebaut haben. Über die Geokoordinaten wird die lokale Zuordnung der Datenübertragung erreicht. Dies wird benötigt um in der Kartendarstellung der Messung einen Ort zuzuweisen. Aus den Messwerten werden die Felder PRES, TEMP und PSAL ausgewählt. Neue Modelle der Messbojen besitzen noch das Feld COND über den sich die Leitfähigkeit des umliegenden Wassers herleiten lässt. Dieser ist aber noch nicht bei vielen Messstationen implementiert und wird deswegen

Datenfeld	Beschreibung	Datentyp
PLATFORM_NUMBER	Eindeutige Identifikationsnummer einer Messstation	<STRING8>
CYCLE_NUMBER	Fortlaufende und innerhalb einer Messboje eindeutige Identifikationsnummer eines Messprofils	<int32>
JULD	In Julian Date codierter Zeitpunkt der Übertragung einer Messung	<float64>
N_PARAM	Anzahl der Messsensoren	<STRING16>
LATITUDE & LONGITUDE	Die Positionsdaten einer Messung zum Zeitpunkt der Übetragung der Werte	<float64>
PRES	Messvektor des Wasserdrucks	<float32>
TEMP	Messvektor der Wassertemperaturen	<float32>
PSAL	Messvektoren des Salzgehaltes	<float32>

Tabelle 1: Beschreibung der ausgewählten Daten

hier nicht verwendet.
Die verwendeten Messwerte PRES, TEMP und PSAL eines Messprofils liegen in vektorieller Form vor. Für die Darstellung über einen univariaten Graphen wird nur ein skalarer Wert pro Messprofil benötigt. Die Daten sollen zusammengefasst werden, bevor diese in die Datenstruktur überführt werden, um die Größe der Daten zu verringern.

3.6 Technische Anforderungen

3.6.1 Verwendete Programmiersprachen

Die Darstellung der Webapplikation wird mit HTML und Javascript umgesetzt. Diese Sprachen gelten in der Entwicklung von Webseiten als Standard und werden von den gängigen Browsern unterstützt. Die weiteren Teile der Applikation soll mit möglichst einer einzigen Programmiersprache entwickelt werden, die alle benötigten Teilbereiche umsetzen kann.
Das Öffnen der netCDF Dateien und die numerische Berechnung kann mit C, Java/Scala, R und Python erfolgen. Insbesondere die beiden letzten sind in der Datenverarbeitung und Numerik als etablierte Werkzeuge zu sehen.
Die Darstellung der Seite soll durch ein Webframework unterstützt werden. Hier gibt es in beinahe allen Hochsprachen entsprechende Werkzeuge. Wählt man aus der Problemstellung der numerischen Verarbeitung R und Python heraus, so ist die Auswahl der geläufigen Webframeworks bei Python höher anzusehen. Diese Sprache besitzt eine große Anzahl von Bibliotheken für die Anbindung von Datenbanken, sowie einige etablierte Bibliotheken für die Schaffung von Webapplikationen.

Python besitzt in Hinsicht auf Laufzeitkosten und einer nicht strikten Typisierung einige Nachteile gegenüber Sprachen wie C und Java. In Hinsicht die auf die Auswahl der Programmbibliotheken, sowie der numerischen Berechnung besitzt überwiegen die Vorteile und wird deswegen hier für die Implementierung genutzt.

3.6.2 Betriebssystem des Servers

Für die Laufzeitumgebung der Applikation wird ein GNU/Linux eingesetzt. Die Software wird so entwickelt, dass sie unter den gängigen Distributionen lauffähig sein wird. Für den Betrieb dieser Applikation wird sich hier für die Distribution Debian-Stable entschieden. Debian ist sehr verbreitet und stark auf Stabilität ausgelegt, sie zeichnet sich insbesondere gegenüber weiteren Distribution mit einer ähnlichen Verbreitung wie RHEL und Ubuntu durch ihre nicht kommerzielle Ausrichtung aus. Als Nachteil ist der fehlende kommerzielle Support zu sehen. Diese Arbeit sollte voraussichtlich aber in keine Bereiche vordringen, welche einen kostenpflichtigen Support rechtfertigen würden.

3.6.3 Webframework

Für Python gibt es eine große Anzahl an Werkzeugen für die Entwicklung von Webseiten. Hier werden exemplarisch 3 herausgegriffen und für die hier vorliegende Aufgabe bewertet.

Django wird beworben als „The web framework for perfectionists with deadlines“. Es wurde 2005 unter einer BSD Lizenz released und entwickelt, um die News-Seite des *Lawrence Journal-World* umzusetzen und zu verwalten. Django ist ein etabliertes und häufig verwendetes Webframework. Es ist dynamisch einsetzbar und für eine große Anzahl von Anwendungen verwendbar. Die Bibliothek ist außerdem durch Module erweiterbar. Die Software folgt dem „*batteries included*“ Ansatz und liefert in der Grundausstattung bereits alles nötige mit, um eine Webseite inklusive Login und Eingabemasken für journalistische Tätigkeiten auszubauen.

Flask ist ein sogenanntes Micro-Framework. Es verwendet die Toolsammlung *Werkzeug* um Webseiten über WSGI darstellbar zu machen. Das Framework folgt dem KISS Ansatz und liefert im Grundumfang nur diejenigen Werkzeuge, die man für die Verwaltung von einfachen Webseiten benötigt. Die Software lässt sich über Module erweitern. Einzelne Teilprojekte von Applikationen lassen sich in sogenannte Blueprints modularisieren.

Falcon ist ebenso als Micro-Framework zu sehen. Es wurde insbesondere in Hinblick auf Geschwindigkeit entwickelt. Das Framework erlaubt Requests asynchron zu verarbeiten und lässt die auf Geschwindigkeit spezialisierte Python-Laufzeitumgebung

pypy zu. Falcon ist ein relativ neues Framework und erfährt in letzter Zeit zur Schaffung von REST-APIs immer mehr Aufmerksamkeit. Es ist aber auch möglich, mit diesem Framework Webapplikationen mit einer Anzeige über HTML-Elementen zu gestalten.

In dieser Applikation wird die Schaffung von Journalistischen und redaktionellen Inhalten eine sehr geringe Rolle spielen. Unter diesem Gesichtspunkt ist die Frage der Komplexität der verwendeten Software zu klären. Unter diesem Aspekt erscheint Django nicht als die ideale Wahl.
Flask und Falcon verfolgen beide den Ansatz eines Microframeworks. Die Laufzeitgeschwindigkeit des Controllers erscheint an dieser Stelle nicht als limitierender Faktor der Applikation. Zwar wäre es wünschenswert, die Datenbeschaffung der Webapplikation bereits im Backend asynchron zu erledigen, doch überwiegt das breitere Spektrum an Modulen und Dokumentationen für die Webentwicklung von Flask.
Damit erscheint Flask als das geeignetste Werkzeug für diese Aufgabe.

3.6.4 Datenbank

Das DBMS soll über Schnittstellen in den Sourcecode des Programmes eingebunden werden. Die datenbank soll einem relationalen schema folgen. Die Verwendung sollte kostenfrei möglich sein und die benötigte Software über die Quellen des betriebssystems verfügbar sein.
In dieser Applikation wurde sich für die Verwendung von PostgreSQL entschieden. Als O/R-Mapper steht für Python SQLAlchemy zur Verfügung.

4 Systementwurf

4.1 Modellierung der Datenbank

Um die Daten zu modellieren ist es sinnvoll, sich diese im Kontext der Erhebung zu betrachten. In (1) ist dieser Prozess vereinfacht dargestellt.

$$\text{Boje} \rightarrow (\text{misst}_{\text{an Ort}}) \rightarrow \text{Messprofile} \tag{1}$$

Dabei ist erkennbar, dass dieses Modell eine Verkettung von Entitäten und Ereignissen darstellt. Eine Boje misst über ihre Lebensdauer eine Anzahl von Messzyklen. Jeder dieser Messzyklen besteht aus einer gewissen Anzahl von Messwerten.
Um das Modell weiter fortzuführen, wurde diese Ereigniskette in ein Entitätsschema überführt. In Abbildung 6 ist die hier verwendete Modellierung des Prozesses zu sehen.

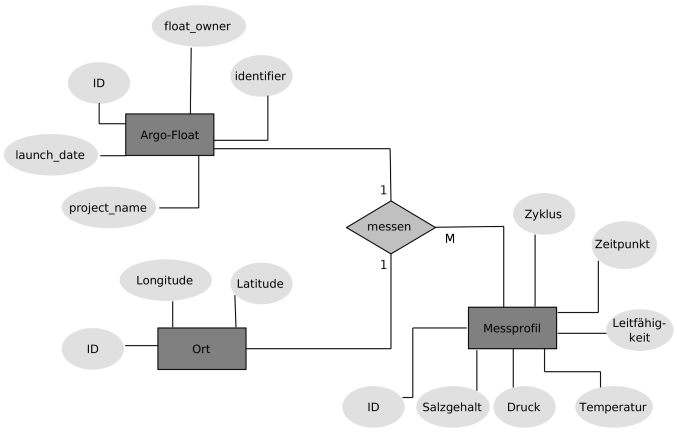

Abbildung 6: Beschreibung der Entitäten der Datenaggregation

4.2 Architektur

Wie in Abbildung 7 zu sehen ist, besteht die Applikation aus zwei Grundkomponenten. Ein Teil ist für die Beschaffung und Aufbereitung der Daten zuständig, während der zweite Teil für die Darstellung der Daten zuständig ist.

4.2.1 Entwurf der Datenaggregation

Die Datenaggregation erfüllt zwei Funktionen. Zum einen muss sichergestellt sein, dass die Daten aus den vom Argo Programm bereitgestellten Strukturen gelesen und in ein logisches Format überführt werden. Des weiteren müssen diese Daten in die Datenbank der Applikation überführt werden.

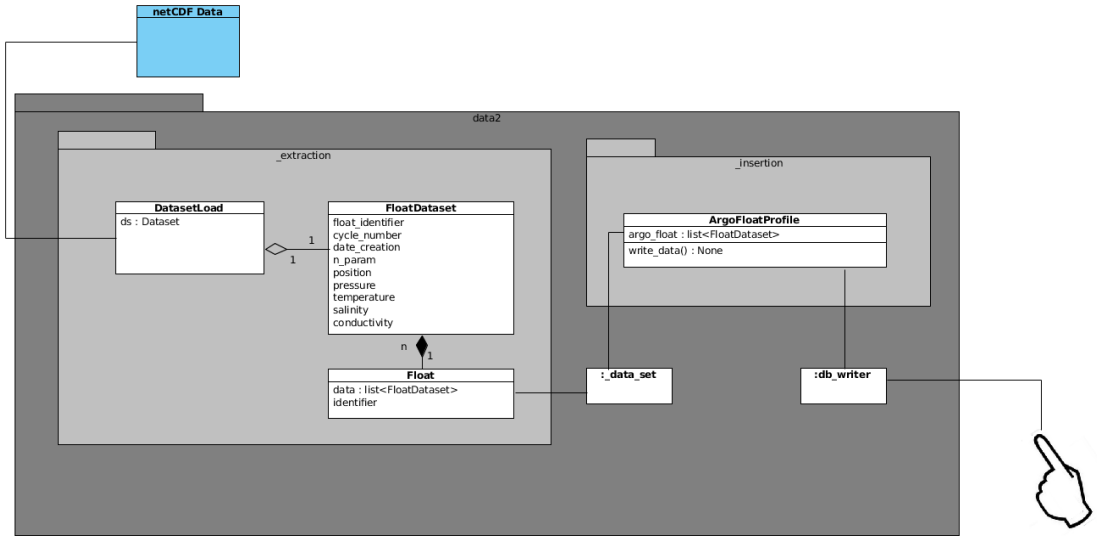

Abbildung 7: Entwurf der Architektur der Aggregation der Daten

Auch in diesem Modell ist die Ereigniskette aus (1) abzubilden. Jede Messstation wird über ein Objekt abgebildet. Die hier gespeicherten Daten sind für eine Messstation eindeutig. Jeder Messzyklus der Boje wird über ein weiteres, zur Messstation gehöriges Objekt abgebildet. Dieses ist innerhalb des Kontextes eindeutig und trägt die Informationen zur jeweiligen Messung.
Für die Persistierung werden objektorientierte Strukturen für den ORM verwendet. Diese Strukturen teilt sich das Modul mit der Webapplikation. Die Behandlung der Daten sowie eine Steuerung des Prozesses sind als weitere Schnittstellen zu definieren.
Die Aggregation der Daten wird über zwei Teilbereiche abgebildet. Zum einen müssen die benötigten Parameter aus den Dateien ausgelesen und modelliert werden.
Als zweite Ebene der Aggregation ist der Prozess des Schreibens in eine Datenbank zu sehen. Diese verwendet die zuvor ermittelten modellierten Messprofile und schreibt sie Anhand der dort enthaltenen Daten in eine Datenbank.
Es ist an dieser Stelle zu erkennen, dass zwei Stellen existieren, die die intrinsischen Eigenschaften des Modules an dieser Stelle beeinflussen. Im folgenden werden die Schnittstellen aufgelistet, die eine richtige Handhabung festsetzen:

Schnittstellen

1. **Daten**

    a) Die Datenstruktur ist normiert. Es ist sicherzustellen, dass die bekannte Daten- bzw. Ordnerstruktur abgearbeitet wird. Mit Änderungen in dieser Struktur muss nicht gerechnet werden.

b) Es muss sicher gestellt werden, dass geöffnete Dateien wieder geschlossen werden.

2. **Steuerung**

   a) Daten separiert in die Datenbank zu schreiben, könnte zu Inkonsistenzen führen und soll vermieden werden.

   b) Der Prozess sollte als Sequenz modelliert werden.

   c) Es muss sicher gestellt werden, dass die Daten schrittweise in die Datenbank überführt werden. Würden zu Beginn alle Dateien geöffnet und gemeinsam im flüchtigen Speicher vorgehalten,· könnte es zu Problemen führen.

4.2.2 Entwurf der Webapplikation

Die Webapplikation besteht aus zwei Teilkomponenten. Die erste Komponente ist für die Darstellung der Webseite zuständig (app). Durch diese werden Elemente in HTML, Javascript und als Bilder ausgeliefert, welche im Webbrowser der Benutzenden angezeigt werden können. Die Darstellung erfolgt dabei dem Singlepage-Prinzip.
Die zweite Teilkomponente ist dafür zuständig, benötigten Daten bereitzustellen (api). Alle Daten aus Datenhaltung werden über diese Schnittstelle angefordert. Die Daten werden über JSON codiert.

4.2.3 Ausarbeitung der Webrouten

Die **api** ist dafür zuständig, die benötigten Daten der Applikation bereitzustellen. Über definierte Webrouten wird über ein GET-Request ein JSON angefordert. Die hierfür ausgearbeitete Struktur ist in Listing 3 zu sehen und ist im Folgenden beschrieben:

```
1 (1.1) GET     /last_seen
2 (1.2) GET     /last_seen/[force_reload]
3 (1.3) GET     /argo_float/[identifier]
4 (1.4) GET     /positions/[identifier]
```

Listing 3: Webrouten der API

(1.1) / (1.2) Über diese Route werden die Daten für die letzte Position der Messstationen aufgerufen. Dieser Datensatz trägt die Daten, die zur Anzeige der Positionen der Messstationen auf der Weltkarte benötigt werden und die Zusatzinformationen zur Darstellung eines Tooltips der einzelnen Argo-Floats. Die Daten werden bei jedem Besuch der Webseite ausgeliefert. Da sich diese erst verändern, wenn neue Daten

vorliegen, werden diese über einen Caching-Mechanismus vorgehalten. Der optionale Übergabeparameter `force_reload` ermöglicht das Neuanlegen des Caches. Um Vandalismus vorzubeugen, sollte es sich dabei um einen nicht zu erratenden Token handeln.

(1.3) Über diese Route werden die Mess-, wie auch Metadaten einer Messstation angefordert. Die Auswahl der Boje erfolgt über den Übergabeparameter `identifier`. Hier wird die eindeutige Identifikationsnummer (siehe PLATTFORM_NUMBER in Tabelle 1) der Messstation verwendet.

(1.4) Um den Positionsverlauf einer Boje anzufordern, dient diese Route. Auch hier wird zur Identifikation der Messstation deren Identifikationsnummer als Parameter übergeben.

Die **app** liefert die Teile aus, die vom Benutzer gesehen werden. Es handelt sich dabei um HTML/Javascript und Bilddaten. Die hierfür ausgearbeitete Struktur ist in Listing 4 zu sehen und im Folgenden beschrieben:

```
1 (2.1) GET     /
2 (2.2) GET     /chart/[identifier]
3 (2.3) GET     /info/[identifier]
```

Listing 4: Webrouten der APP

(2.1) Über diese Seite ist die eigentliche Applikation zu sehen. Dies umfasst die Karte sowie die angezeigten Messwerte.

(2.2) Diese Route stellt den Plot der darzustellenden Messwerte als Bild zur Verfügung.

(2.3) Über diese Route wird ein HTML für die Infoanzeige einer Messstation bereitgestellt.

4.2.4 Aussehen der Webapplikation

Das Aussehen der Applikation soll dem Muster von gängigen Webapplikationen entsprechen. In Abbildung 8 ist ein erster Grobentwurf der Applikation zu sehen.
Zentrales element der Applikation ist die Darstellung einer Karte. Über diese sollen die letzten Positionen der Karte ersichtlich sein.
Klickt man eine Messstation an, so wird auf der linken Seite ein weiteres Element in die Seite eingefügt. Dieses dient zur Darstellung der Messwerte des jeweiligen Argo-Floats.

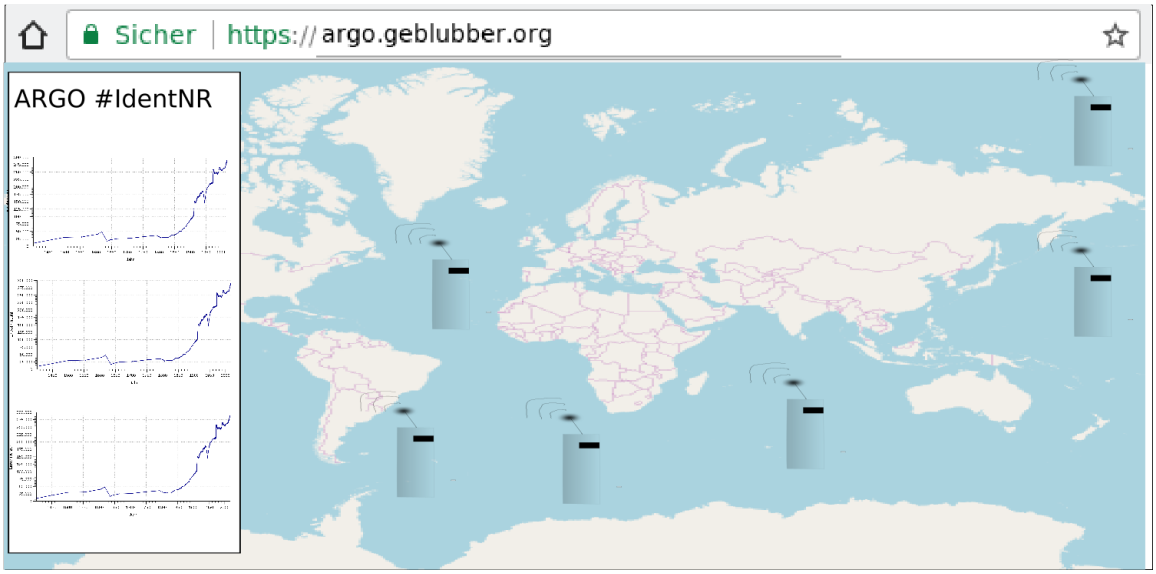

Abbildung 8: Grafischer Grobentwurf der Webapplikation

5 Implementierung

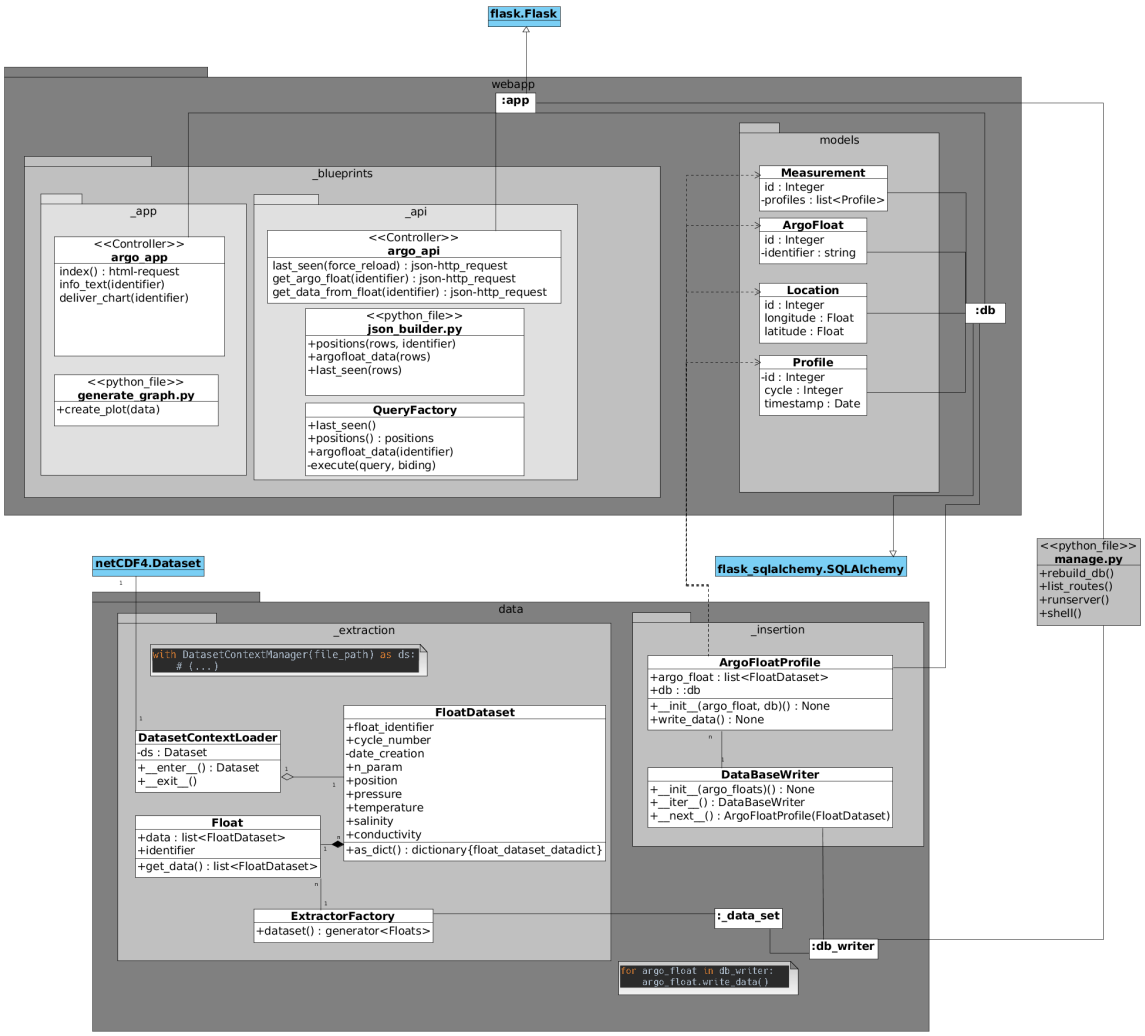

Abbildung 9: Architekturbeschreibung von ArgoData

5.1 Datenaggregation

5.1.1 Auslesen der Daten

Der Datensatz besteht vor der Verarbeitung aus einer Ansammlung von Verzeichnissen und Dateien. Da die Gefahr besteht, dass geöffnete Dateien nicht wieder ordnungsgemäß geschlossen werden, muss eine Schnittstelle geschaffen werden, die unsachgemäße Verwendung der Dateien verhindert. Python sieht für diesen Zweck den Kontextmanager vor. In Listing 5 ist die hier verwendete Implementierung zu sehen.

```
from netCDF4 import Dataset

class DatasetContextManager(object):
    def __init__(self, file_path):
        self.ds = Dataset(file_path)

    def __enter__(self):
        return self.ds

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.ds.close()
```

Listing 5: Kontextmanager zur Sicherstellung der richtigen Dateibehandlung.

In Listing 6 ist die Verwendung des Kontextmanagers aufgezeigt und im Folgenden beschrieben.

```
with DatasetContextManager(file_path) as ds:
    juld = ds.variables['JULD'][0]
```

Listing 6: Die Verwendung des Kontextmanagers

Durch die Verwendung der Struktur über das Schlüsselwort `with` wird die Objektmethode `__enter__` aufgerufen. Diese gibt den objekteigenen netCDF-Datensatz zurück. Dieses wird in der Verwendung im gleichnamigen Objekt `ds` gespeichert. Im darauf folgenden Operationsblock kann das Objekt nun verwendet, und dessen Daten extrahiert werden. Nachdem der Operationsblock verlassen wurde, wird die Methode `__exit__` des Kontextmanagers aufgerufen. Innerhalb dieser methode wird die Datei nun geschlossen. Durch dieses Entwurfsmuster ist die Richtige Verwendung der Datei also transparent sichergestellt. Ein Entwickler kann somit nicht durch Unachtsamkeit vergessen, den Datensatz zu schließen. Gleichzeitig erhöht sich die Lesbarkeit. Durch den Operationsblock ist jederzeit ersichtlich, ab welcher Stelle im Quelltext der Datensatz nicht mehr verfügbar ist.
Diese Schnittstelle wird in den Objektrepräsentationen der Argo-Floats und der Datensätzen verwendet, um auf die Daten zuzugreifen. In diesen wird wie in Kapitel 4.2.1 gezeigt, die Datenstruktur vorgehalten.

5.1.2 Schreiben der Daten

Um die Verarbeitung der Daten sowie den Prozess der Aggregation in der Datenbank richtig zu modellieren, ist es sinnvoll, den Prozess als Sequenz zu modellieren. Dies erlaubt es, die Datenstruktur über einen Generator vorzuhalten. Python sieht dafür den Iterator vor. Die hier verwendete Implementierung ist in Listing 7 zu sehen. Das Interface wird über die Funktion `def __next__(self)` realisiert. Dieses delegiert die Iteration zum Objekteigenen Generator `self.argo_floats`. In dem Moment, in dem ein Objekt aus dem Generator

angefordert wird, findet die Verarbeitung der Datenstruktur statt. Zum Schluss wird der Datensatz über ein Objekt ausgeliefert, das es erlaubt, die Daten in die Datenbank zu überführen. Damit ist sichergestellt, dass der Prozess nur als Sequenz verwendet werden kann.

```
from ._argo_float_profile_writer import ArgoFloatProfile


class DataBaseWriter:
    def __init__(self, argo_floats, db, app):
        self.argo_floats = argo_floats
        self.db = db
        self.app = app

    def __iter__(self):
        return self

    def __next__(self):
        self.argo_float = next(self.argo_floats)
        return ArgoFloatProfile(self.argo_float, self.db, self.app)
```

Listing 7: Implementierung des Iterators zur Steuerung der Aggregationssequenz

Die Verwendung der Schnittstelle ist in Listing 8 aufgezeigt.

```
from data import db_writer

for argo_float in db_writer:
    argo_float.write_data()
```

Listing 8: Verwendung der Schnittstelle zur Steuerung der Datenaggregation

Eine zentrale Problemstellung im Prozess der Überführung der Daten in das Relationale Schema ist die effektive Verwendung des flüchtigen Speichers. Eine klassische, sequenzielle Verarbeitung würde die Daten initial auslesen und diese in Gänze im Arbeitsspeicher vorhalten bevor diese über den Mapper in die Datenbank überführt werden. Dieser Prozess wurde hier durch die Verwendung von Generatoren aufgebrochen. In Listing 9 ist die Implementierung dieser Schnittstelle zu sehen. Da in der Comprehension runde statt eckige Klammern verwendet werden, wird initial nur die logische Struktur als Sequenz vorgehalten. Die inhärenten Float-Objekte des Generators werden zu dem Zeitpunkt erzeugt, wenn diese durch eine Iteration aufgerufen werden. Damit werden die Daten erst zu dem Zeitpunkt ausgelesen, wenn diese für die weitere Verarbeitung benötigt werden.
Die Methode `get_data_sets()` erzeugt aus jedem Unterordner im definierten Arbeitsverzeichnis ein Float-Objekt und gibt dieses über das Schlüsselwort `yield` zurück. Durch diese Klasse wird somit ein Generator definiert, der es erlaubt, alle im Arbeitsverzeichnis definierten Argo-Float-Datenobjekte bereitzustellen, ohne diese bereits bei der Instantiie-

rung kennen und abarbeiten zu müssen.

```
import os

from ._float import Float


class ExtractorFactory:
    def __init__(self, data_directory):
        self.extractor_generator = (
            Float(os.path.join(data_directory,
                               profile_directory))
            for profile_directory in os.listdir(data_directory))

        self.__sum_floats = sum([os.path.isdir(os.path.join(
    ↪ data_directory, d)) for d in os.listdir(data_directory)])

    def get_data_sets(self):
        return self.extractor_generator

    def float_count(self):
        return self.__sum_floats
```

Listing 9: Factory zur Extraktion der Datensätze

5.2 Webapplikation

5.2.1 Objektrelationales Mapping

Als Objektrelationalen Mapper wird SQLAlchemy eingesetzt. Diese Software gilt als erprobt und wird bereits in einer Vielzahl von Softwaresystemen eingesetzt. So verwendet unter anderem reddit oder die Mozilla Foundation SQLAlchemy als Schnittstelle zur Datenhaltung. SQLAlchemy wird für die Verwendung in Flask empfohlen und es existiert eine Erweiterung um die Verwendung des Mappers in Flask zu vereinfachen (vgl. [Ron11] S. 33).

In SQLAlchemy werden die Tabelleneinträge in Modellklassen abgebildet. In Listing 10 ist die Implementierung des Models für eine ArgoFloat-Messtation zu sehen.

```
from . import db


class ArgoFloat(db.Model):
    __tablename__ = 'argo_floats'

    measurements = db.relationship('Measurement', backref='argo_floats',
    ↪  lazy='dynamic')

    id = db.Column(db.Integer, primary_key=True)
    identifier = db.Column(db.String(10))
    project_name = db.Column(db.String(100))
```

```
12      launch_date = db.Column(db.Date)
13      float_owner = db.Column(db.String(100))
14
15      def __init__(self, identifier, project_name, launch_date,
        ↪ float_owner):
16          self.identifier = identifier
17          self.project_name = project_name
18          self.launch_date = launch_date
19          self.float_owner = float_owner
20
21      def __repr__(self):
22          return f'<Argo Float {self.id!r}>'
23
24
25  class ArgoFloatTmpTable(ArgoFloat):
26      __bind_key__ = 'data_input'
```

Listing 10: Modellklasse für ein Argo-Float

Die Registrierung des Models erfolgt über die Vererbung der Metaklasse db.Model. Die Eigenschaften der jeweiligen Entitäten werden über Attribute des Objektes implementiert. Diese werden in Instanzen von db.Column transferiert. Dies erlaubt das Festsetzen des Datentyps. So lassen sich auch weitere Datenspezifikionen definieren. Die Attribute werden durch eine Parameterübergabe in den Initiator der Klasse mit Werten versehen. Um das Binding zu umzusetzen, wurden Nachfahren der Modellklassen erzeugt. Diese benötigen für die Zuordnung das Attribut __bind_key__.
Über das Schlüsselwort db.relationship werden Beziehungen beschrieben. In diesem Fall besteht eine $1 - N$ Beziehung zu measurements(argo_float ← measurements) (Siehe auch Abbildung 6). Als Übergabeparameter wird ein Name für die Beziehung erwartet. Der Parameter backref gibt den Namen der Klasse auf dem Mapper an. Diese Einstellung erlaubt den Aufruf der Beziehung auch in die andere Richtung. Der Parameter lazy definiert die verwendete Strategie für das Lazyloading. In diesem Fall wurde sich für 'dynamic' entschieden. Diese Einstellung gibt bei Lesezugriffen ein vorkonfiguriertes Query-Objekt zurück. Dies erlaubt das Hinzufügen weiterer Filter vor dem Zugriff auf die Tabellen. Die durch diese Beziehung verbundene Modellklasse Measurements ist in Listing 11 zu sehen.

```
1   from . import db
2
3
4   class Measurement(db.Model):
5       __tablename__ = 'measurements'
6
7       id = db.Column(db.Integer, primary_key=True)
8
9       profiles = db.relationship('Profile', backref='measurements', lazy='
        ↪ dynamic')
10
11      argo_float_id = db.Column(db.Integer, db.ForeignKey('argo_floats.id'
```

```
    ↪ ))
    argo_float = db.relationship('ArgoFloat')

    location_id = db.Column(db.Integer, db.ForeignKey('locations.id'))
    location = db.relationship('Location')

    def __init__(self, argo_float, location):
        self.argo_float = argo_float
        self.location = location

    def __repr__(self):
        return f'<Measurement {self.id!r}>'


class MeasurementTmpTable(Measurement):
    __bind_key__ = 'data_input'
```

Listing 11: Modellklasse für eine Messung

Auf dieser Seite der Beziehung (measurement → argo_float) unterscheidet sich deren Implementierung. Für die eindeutige Zuordnung des übergeordneten Argo-Floats wird deren id als Fremdschlüssel registriert. Die Beziehung benötigt neben dem Namen der anderen Seite keine weiteren Parameter, da die Beziehung bereits konfiguriert worden ist. Das Schreiben von Daten über den Mapper SQLAlchemy ist Teil der Datenaggregation. Der dort implementierte Programmcode ist in Listing 12 zu sehen.

```
from webapp.models import ArgoFloat, Location, Measurement, Profile


class ArgoFloatProfile:
    def __init__(self, argo_float, db, app):

        self.argo_float = argo_float
        self.db = db
        self.app = app

    def write_data(self, bind=None):
        session = self.db.create_scoped_session(
            options={
                'bind': self.db.get_engine(self.app, bind),
                'binds': {}
            }
        )
        try:
            argo_float_ = ArgoFloat(
                identifier=self.argo_float.identifier,
                project_name=self.argo_float.project_name,
                launch_date=self.argo_float.launch_date,
                float_owner=self.argo_float.float_owner

            )
            for profile_data in self.argo_float.data:
                location_ = Location(
                    latitude=profile_data.position['latitude'],
                    longitude=profile_data.position['longitude']
                )
```

```
31                  measurement_ = Measurement(
32                      argo_float=argo_float_,
33                      location=location_
34                  )
35
36                  session.add(Profile(
37                      cycle=int(profile_data.cycle_number),
38                      timestamp=profile_data.date_creation,
39                      measurement=measurement_,
40                      salinity=profile_data.salinity,
41                      pressure=profile_data.pressure,
42                      temperature=profile_data.temperature,
43                      valid_data_range=profile_data.valid_data_ranges
44                  ))
45
46              session.commit()
47          except Exception as err:
48              session.rollback()
49              raise err
```

Listing 12: Das Schreiben der Daten eines Argo-Floats in die Datenbank

Die hier implementierte Klasse `ArgoFloatProfile` ist für das Schreiben aller Datensätze eines ArgoFloats zuständig. Diese verwendet für die Zuordnung der Daten die Modell-Klassen aus der Webapplikation. Durch die Methode `write_data` werden die Daten des durch `__init__` übergebenen Datensatzes in die Datenbank überführt. Um ein dynamisches Binding realisieren zu können, wird der betreffende Parameter `bind` übergeben. Mithilfe dieses Parameters wird eine Session aufgebaut, welcher das Schreiben in die Datenbank erlaubt, die über das binding definiert ist.
Für jeden Datensatz wird eine Instanz der betreffenden Modell-Klasse erstellt. Die dafür benötigten Daten werden aus den Datensätzen des Klasseneigenen Generator-Objektes `argo_float` angefordert und als Parameter übergeben.
Der zuvor definierten Session werden nur die Instanzen der `Profile`-Klassen übergeben. Da alle zum Messprofil gehörenden Modells in diesem Kontext eindeutig sind, kann SQLAlchemy diese selbstständig zuordnen. Zum Abschluss werden die Daten über `session.commit` `()` in die Datenbank überführt. `session.rollback()` erlaubt es, die Veränderungen, die innerhalb dieser Session an der Datenbank herbeigeführt wurden, wieder auf den Urzustand zurückzuführen.

SQLAlchemy bietet eine eigene Abfrage-Sprache an. In Listing 13 ist eine vereinfachte Implementierung der Abfage zur Bestimmung der Positionshistorie einer Argo-Float zu sehen, wie sie auch in der Web-API implementiert ist.

```
1 from webapp import db
2 from webapp.models import Measurement, Location, Profile, ArgoFloat
3
4 query = db.session.query(ArgoFloat, Location, Profile) \
```

```
                .join(Measurement) \
                .join(Location) \
                .join(Profile) \
                .filter(ArgoFloat.identifier == '1900037') \
                .order_by(Profile.timestamp)

result = db.session.execute(query, None, bind=db.get_engine(app, None))

data = [row for row in result]
```

Listing 13: Anfragen über SQLAlchemy zum Lesen von Daten

In diesem Beispiel werden über eine Projektion auf Attribute von `ArgoFloat`, `Location` und `Profile` Daten angefordert. SQLAlchemy erlaubt die Definition von Joins über mehere Tabellen. Diese werden durch die in den Modell-Klassen definierten Beziehungen aufgelöst. Die Selektion erfolgt durch die Methode `query.filter()`. In diesem Fall werden nur Relationen der Argo-Float mit dem identifier '1900037' ausgewählt. Die Methode `query.order_by()` erlaubt das Sortieren der Ergebnisse. In diesem Fall werden die Datensätze durch das Attribut `Profile.timestamp` chronologisch sortiert.

SQLAlchemy ist auch in der Lage, in SQL-Sprache definierte Anfragen zu verarbeiten. In Listing 14 ist eine vereinfachte Implementierung aus der Web-API zu sehen. Diese extrahiert die letzten Positionen mit den dazugehörigen Zeitpunkten aus der Datenbank.

```
from webapp import db

sql_template = """SELECT DISTINCT ON (f.id)
  f.identifier,
  l.longitude,
  l.latitude,
  p.timestamp
FROM argo_floats f
  JOIN measurements m
    ON f.id = m.argo_float_id
  JOIN profiles p
    ON m.id = p.measurement_id
  JOIN locations l
    ON l.id = m.location_id
ORDER BY f.id, p.cycle DESC;"""

result = db.engine.execute(sql_template)
```

Listing 14: Das Ausführen von SQL Anfragen über SQLAlchemy

Über die Methode `db.engine.execute()` können SQL-Queries direkt verarbeitet werden. Dies hat aber zwei entscheidende Nachteile. Der Query-Dialekt beschränkt die Anfrage auf ein bestimmtes DBMS. Diese Anfrage ist für PostgreSQL entworfen und würde auf einem anderen DBMS nicht funktionieren. Zum anderen würden Eingaben nicht gegen SQL-Injections gesichert werden. Eine Abfrage wie `db.engine.execute(f'SELECT *`

↪ `FROM argo_floats WHERE argo_floats.identifier={user_input})` würde also unabschätzbare Sicherheitsrisiken mit sich bringen.

Flask bietet für die Einbindung von SQLAlchemy eine Erweiterung an. Durch diese Erweiterung wird eine SQLAlchemy-Instanz durch ein zentrales und scheinbar globales Objekt (db) repräsentiert. Dies ermöglicht eine einfache Datenabstraktion und ist für Flask typisch. Dieser Mechanismus wird durch die zeitkritische und in ihrem Ablauf fest vorgeschriebene Erstellung von Objekten und Submodulen erkauft. Dieser Prozess ist bei der Einbindung von `flask-sqlalchemy` gut sichtbar. Aus diesem Grund wird die Initialisierung dieser Erweiterung hier gesondert dargestellt.
In Abbildung 5.2.1 ist eine sequentielle Darstellung des Prozesses zu sehen und wird im Folgenden beschrieben.

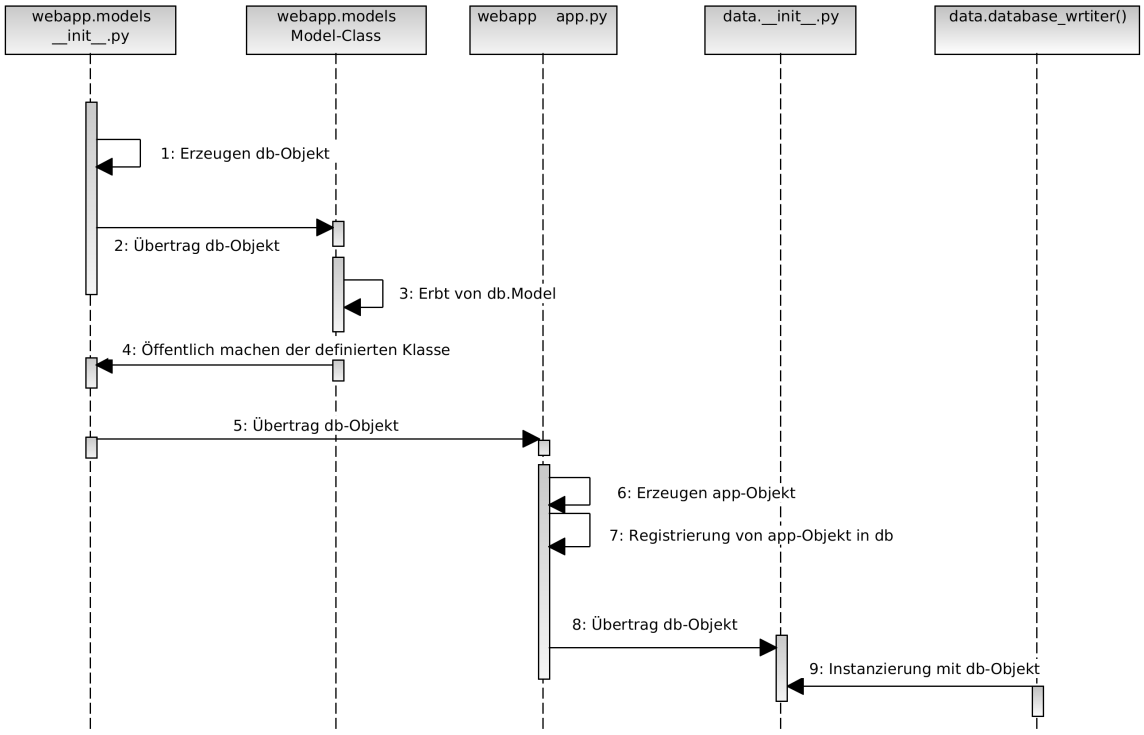

Abbildung 10: Sequenzdiagramm der Einbindung von SQLAlchemy

1. In der Datei `webapp.models.__init__.py` muss sichergestellt werden, dass die SQLAlchemy-Instanz noch vor der Initiierung der Modell-Klassen erzeugt wurde.

2. Nun kann das db-Objekt in die Modellklassen übertragen werden. Dafür importieren diese db in ihren Scope.

3. Die Modell-Klassen werden in db registriert. Dafür erben diese von der Metaklasse

db.Model. Siehe auch Listings 10 und 11.

```
1 from flask_sqlalchemy import SQLAlchemy
2
3 db = SQLAlchemy()
4
5 from ._measurement import Measurement
6 from ._argo_float import ArgoFloat
7 from ._location import Location
8 from ._profile import Profile
```

Listing 15: webapp.models.__init__.py

4. Die Modell-Klassen dürfen erst nach der Instantiierung von db über den Scope des Submodules heraus bekannt gemacht werden, um sicherzustellen, das dieses bereits existiert. Die Implementierung des Ablaufs ist in Listing 15 ersichtlich.

5. Die Instanz db kann nun mit den in ihr registrierten Modell-Klassen aus dem Submodul in die Datei webapp.__init__.py importiert werden.

6. Hier wird nun das globale Objekt der Flask-Instanz (app) erstellt und konfiguriert.

7. Anschließend wird app in db registriert. Die Schritte 5 bis 7 können über das Listing 16 nachvollzogen werden.

8. Die Instanz db aus dem Scope webapp.db ist nun fertig konfiguriert und kann für das Lesen und Schreiben in die Datenbank verwendet werden.

5.2.2 Controller

```
1  from flask import Flask, url_for
2  from flask_twisted import Twisted
3
4  from .models import db
5
6  app = Flask(__name__, static_url_path='/static')
7  app.config.from_pyfile('./argo.cfg')
8
9  db.init_app(app)
10
11 from .blueprints import argo_api, argo_app
12
13 app.register_blueprint(argo_api)
14 app.register_blueprint(argo_app)
15
16 from . import models
17
18 twisted = Twisted(app)
```

Listing 16: webapp.__init__.py

Über die globale Instanz des Controllers (app) kann auf Daten zugegriffen und die Konfiguration vorgenommen werden. In Listing 16 ist die Initialisierung des Controllers zu sehen.
Über Blueprints wird die in Kapitel 4.2.3 ausgearbeitete logische Trennung zwischen app und api realisiert. Die hier definierten Module umfassen auch die in diesem Kontext benötigten Hilfsprogramme. Die hier implementierte Modulstruktur von argo_api und argo_app ist im Folgenden ersichtlich.

argo_api

query_factory Alle lesenden Zugriffe zur Datenbank werden über eine QueryFactory zentral zusammengefasst. Die hier ausgearbeitete Implementierung Ist in Kapitel 5.2.1 ausführlich beschrieben.

json_builder Durch diese Programmabschnitte werden die Daten in Listen und Dictionaries überführt. Die Variablen und Datenstrukturen werden dabei so angelegt, dass diese direkt als JSON bzw geoJSON ausgeliefert werden können.

Dieses Controller-Blueprint fordert die Daten über die query_factory an und bringt sie über den json_builder in das vorgesehene Format. Die Umwandlung in JSON und die Erstellung des Requests erfolgt über flask.jsonify.

argo_app

generate_graph Dieser Programmteil erzeugt einen Plot über die Bibliothek `matplotlib`. Eine nähere Beschreibung der Erstellung findet sich in Kapitel 5.2.5.

In diesem Blueprint werden Templates zu HTML-Dateien zusammengesetzt. Dafür benötigt dieser Programmteil Zugriff auf die Verzeichnisse für die jeweiligen Templates und statischen Dateien. Diese Daten werden über `flask.render_template` zusammengesetzt und ausgeliefert.

Die so erstellten Blueprints werden im zentralen Controller (app) über das Schlüsselwort `app.register_blueprint` registriert und zusammengefasst.
Als zentrale Steuerungseinheit für den gesamten Programmablauf dient die Datei `manage.py`. Die Parameter dieser Datei werden im Folgenden kurz beschrieben

runserver Durch diesen Parameter wird der Server der Applikation gestartet.

rebuild_db Dies stößt den Prozess der Datenaggregation an. Streng genommen ist dies nicht Teil des Controllers. Aus Gründen der Einfachheit wurde dieser Programmablauf aber in die Steuerungseinheit integriert.

shell Durch diesen Parameter wird eine IPython-Shell gestartet. In diesem repl ist die Webapplikation mit all ihren Umgebungsvariablen initiiert. Diese Shell dient für das Debugging der Applikation.

list_routes Dieser Parameter listet alle Routen, die durch den Controller definiert sind auf der Kommandozeile auf.

5.2.3 Templates

Für das Erstellen der HTML-Dateien wurde die Template-Engine Jinja2 verwendet. Diese ist gut in Flask integriert. Jinja2 erlaubt das Modularisieren von HTML Dateien, den Zugriff auf Variablen und übergebenen Daten sowie die Abarbeitung einfacher Strukturen und Wahrheitswerte.
Die Templatestruktur wurde in dieser Applikation stark modularisiert. In Listing 17 ist der Einstieg in die Modulstruktur zu sehen und im Folgenden beschrieben.

```
1 {% extends '_base.html' %}
2
3 {% block map %}
4     {% include '_render_map.html' %}
5 {% endblock %}
```

Listing 17: `webapp.templates.map.html`

Die Datei _base.html ist für die Darstellung der Webseite verantwortlich. Diese wurde über Twitter Bootstrap realisiert. Als Vorlage diente dabei ein bereits ausgearbeitetes 2-Column Layout aus [Ng14]. Der Beispielcode wurde an die Anforderung der Applikation angepasst. Der Code wurde dabei in die logischen Bestandteile zerlegt und in die modulare Struktur der Webseite überführt. Der Code ist unter einer MIT Lizenz veröffentlicht. Damit kann der Code unter Nennung der Lizenz verwendet werden. Deswegen wurde ein Tag in das HTML Element eingesetzt, um Autor und Lizenz zu nennen.

5.2.4 Kartendarstellung

OpenLayers ist eine Programmbibliothek um interaktive Geoapplikationen zu entwickeln. Das Framework ist in Javascript entwickelt und nimmt alle benötigten Berechnungen auf Clientseite vor. Die Bibliothek erlaubt es Kartenmaterial aus verschiedensten Quellen zu rendern. Dabei können Kachel- oder auch Vektorbasierte Materialien eingebunden werden. Die Elemente der Karte setzen sich wie in Listing 18 zu sehen zusammen.

```
1  var map = new ol.Map({
2     layers: [mapVectorLayer, argoFloatsLayer],
3     target: 'map',
4     controls: ol.control.defaults({
5         attributionOptions: false,
6         zoom: false,
7     }),
8     view: new ol.View({
9         center: [0, 0],
10        zoom: 3.5,
11        minZoom: 3
12    })
13 });
```

Listing 18: Das ol.Map Element aus der Kartendarstellung

Die Darstellung bassiert auf zwei Layern, einer Verktordarstellung der Kontinent- und Landesgrenzen (mapVectorLayer), sowie der Darstellung der ArgoFloats (argoFloatsLayer). Die Darstellung der Steuereinheiten wurde deaktiviert. Als Startposition wurden Längen- und Breitengrad mit jeweils 0 gewählt. Die Zoomstufe ist mit 3 initialisiert und es ist eine maximale Zoomstufe von 3.5 gewählt. Die Werte für die Zoomstufe wurden bei einer Auflösung von 1920x1080 ermittelt und getestet. Die Karte bettet sich in das HTML-Element map ein und wird mit diesem zur Anzeige gebracht. Steuergesten mit Maus und Tastatur nimmt die Karte an dieser Stelle bereits entgegen.
Der mapVectorLayer wurde aus der GeoJSON-Datei countries.geo.json erstellt. Die Datei wurde von [Sun16] heruntergeladen. Diese steht unter der Lizenz UNLICENSE und kann damit ohne Einschränkungen verwendet werden. Die Darstellung der Argo-Floats wird ebenso über eine GeoJSON realisiert. Diese wird auf dem Webserver über die API

bereitgestellt und wird über der URL `/last_seen` (siehe. Listing 3) angefordert. Die so angeforderte Datenstruktur hat ein spezifisches Format. In Listing 19 ist das Format in einem gekürzten Format ersichtlich.

```
 {
"crs": {
    "properties": {
      "name": "EPSG:4326"
    },
    "type": "name"
  },
  "features": [
    {
      "geometry": {
        "coordinates": [
          -52.02875,
          12.92591
        ],
        "type": "Point"
      },
      "properties": {
        "feature_type": "latest_position",
        "identifier": "1901673",
        "last_seen": "Sun, 10 Feb 2013 00:00:00 GMT"
      },
      "type": "Feature"
    },
    // (feature_2, ..., feature_n)
    ]
}
```

Listing 19: Gekürzte geoJSON zur Darstellung der Argo-Floats

Der Parameter `crs` (Coordinate Reference System Objects) bestimmt die Art der Kartenprojektion. Durch das property `EPSG:4326` wurde der GPS-Standard WGS84 - World Geodetic System 1984 eingesetzt. Dies deckt sich auch mit den durch die Messstationen bereitgestellten Geo-Koordinaten. Für die Darstellung werden diese Koordinaten in eine Merkatordarstellung (EPSG:3857) projiziert.
Die Repräsentationen der Messstationen finden sich im Array `features`. Im Feld `geometry` werden Position und die Form des Features eingestellt. Über `properties` werden die zusätzlichen Daten `feature_type`, `identifier` und `last_seen` in diesem Format gespeichert. Openlayers kann ein derartig formatiertes GeoJson selbstständig zur Anzeige bringen. Wie die Datei geladen wird, ist in Listing 20 zu sehen.

```
var argoFloatsLayer = new ol.layer.Vector({
    style: FloatStyle,
    source: new ol.source.Vector({
        format: new ol.format.GeoJSON(),
        url: "/last_seen"
    })
});
```

Listing 20: Die Funktion argoFloatsLayer

Als Quelle für die Anzeigedaten wird hierbei ein `ol.source.Vector`-Objekt gewählt. Das Format und die `url` für die spezifizierte Datenquelle werden als Parameter übergeben. Damit werden die ArgoFloats auf einer vektorbasierten Karte angezeigt.
Was zu diesem Zeitpunkt noch fehlt, ist die Möglichkeit mit den ArgoFloats zu interagieren. Bei Hovern mit des Zeigers über einer ArgoFloat soll ein Tooltip mit einigen Informationen erscheinen und die Messstation optisch hervorgehoben werden. Die Hervorhebung wird durch den Befehl `map.addInteraction(hoverInteraction)` in der Karte registriert. `overInteraction` ist eine Instanz von `ol.interaction.Select` und erlaubt die Auswahl von Features und des Event-Typs. In diesem Fall werden alle Features des `argoFloatsLayer` ausgewählt, die unter dem Mauszeiger liegen. Openlayers wählt mit dieser Konfiguration einen Vergrößerungseffekt um das Feature hervorzuheben.
Um den Tooltip zur Anzeige zu bekommen, wurde eine weitere Funktion definiert. In Listing 21 ist die Funktion für das Fangen eines `pointermove`-Events zu sehen.

```
1 map.on('pointermove', function (evt) {
2     if (evt.dragging) {
3         info.tooltip('hide');
4         return;
5     }
6     displayFeatureInfo(map.getEventPixel(evt.originalEvent));
7 });
```

Listing 21: Das Abfangen eines pointermove-Events

Die Funktion wartet auf definierte Eingaben. In diesem Fall wird der Funktionskörper durch das bewegen des Mauszeigers über der Kartendarstellung ausgelöst. Der Event wird als `evt` in die innere Funktion überführt. Dort wird überprüft ob es sich um einen dragging-event handelt. Ist dies der Fall, wird der DIV-Container mit der id `#info` (`info`) ausgewählt und der darin liegende Tooltip über den CSS-Parameter `hide` versteckt. Diese Einstellung wurde gewählt, um störende Tooltips bei der Navigation mit der Karte zu unterbinden. In jeden anderen Fall wird der Pixel an der Spitze des Mauszeigers extrahiert und der Funktion `displayFeatureInfo` übergeben.
`displayFeatureInfo` positioniert das info-Element in der Nähe des Mauszeigers und extrahiert etwaige unter dem Pixel liegende Features. Anschließend werden die Attribute des Features untersucht. Über den Befehl `feature.get("feature_type") === 'latest_position'` werden Features aus dem `last_seen`-GeoJson identifiziert. Ist die Identifikation positiv, werden weitere Attribute extrahiert und in den Text des Tooltips überführt. Zum Abschluss wird der Tooltip über seine CSS-Attribute zur Anzeige gebracht.

Die Funktion kann auch die Features der Positionshistorie einer Boje über einen Tooltip anzeigen. Da sich die hier darzustellenden Parameter unterscheiden, werden dessen Daten gesondert abgefragt.

Die Interaktion über einen Klick passiert analog. In Listing 22 ist die hier verwendete Implementierung der Klick-Event-Abfrage gezeigt.

```
1 map.on('click', function (evt) {
2     displayArgoData(map.getEventPixel(evt.originalEvent));
3     displayPositions(map.getEventPixel(evt.originalEvent))
4 });
```

Listing 22: Das Abfangen eines Klick-Events

Auch an dieser Stelle wird die `map.on`-Funktion der Karte verwendet, um den Event abzufangen. Ist dieser vom Typ `'click'` so werden die darauf folgenden Funktionen mit dem Pixel an der Position des Mauszeigers aufgerufen. Im Folgenden wird er Programmablauf der hier aufgerufenen Funktionen erläutert:

displayArgoData Diese Funktion ist für die Darstellung der Messdaten, sowie des Informationstextes der ausgewählten ArgoFloat verantwortlich. Aus diesem Grund besteht sie aus zwei Unterfunktionen.

display_chart Diese Funktion Stellt den Funktionsplot der Messwerte der ausgewählten ArgoFloat dar. Dafür wird am Anfang der div-Container mit der ID `#chart-picture` ausgewählt. In diesem Container wird später das Bild platziert. Um bereits bestehende Bilder zu löschen, werden alle HTML Elemente inerhalb des Containers erstellt. Anschließend wird das Bild der Messdaten über die Webroute /chart/[identifier] (Siehe Listing 3) angefordert. Das Bild wird anschließend auf die Breite des Containers gebracht und der Bootstrap-Klasse `img-responsive` hinzugefügt. Damit wird das Bild formatiert über die Sidebar dargestellt.

display_info Diese Funktion stellt einen Info-Block zu den Metadaten der ArgoFloat dar. Analog zum oberen Prozess werden hier Container aus dem DOM-Tree selektiert und das sich darin befindliche HTML manipuliert. Die Daten wurden hier als HTML-Objekte über einen Ajax-Befehl angefordert und direkt in den umliegenden Container injiziert.

Die oben genannten Funktionen werden aufgerufen, wenn es sich bei dem angeklickten Element um ein Feature vom Typ `Point` handelt. Der Bereich des Graphen wird an dieser Stelle sichtbar gemacht und der Info-Bereich eingeklappt. Die Sichtbarkeit der Sidebar wird ebenso an dieser Stelle eingestellt.

displayPositions Diese Funktion stellt eine Positionshistorie einer spezifischen Messstation dar. Dafür wird ein GeoJSON über die url `/positions/[identifier]` angefordert. Das sichtbar machen, der hier kodierten Daten geschieht analog zu der Darstellung aller Argo-Floats. Das GeoJson trägt zusätzlich noch ein `LineString`-Element, um den zeitlichen Verlauf der Positionen zu verdeutlichen.

5.2.5 Zeichnen der Graphen

Für die Darstellung der Messwerte wurde sich in dieser Applikation für die Python-Bibliothek `matplotlib` entschieden. Diese Plotting-Engine ist in der Entwicklung von wissenschaftlichen Applikationen in Python sehr verbreitet. Als Renderer wurde ein AGG-Renderer eingesetzt. Dieser erlaubt das Vorhalten der Bilddaten im Arbeitsspeicher, sowie deren Überführung in ein Canvas-Element.
In Listing 23 ist die Funktion des Controllers zu sehen, über die das Bild bereit gestellt wird.

```
1  @argo_app.route("/chart/<identifier>")
2  def deliver_chart(identifier):
3      url = url_for('argo_api.get_argo_float', identifier=identifier,
       ↪ _external=True)
4      data = requests.get(url).json()
5
6      fig = create_plot(data)
7      canvas = FigureCanvas(fig)
8      png_output = BytesIO()
9      canvas.print_png(png_output)
10     response = make_response(png_output.getvalue())
11     response.headers['Content-Type'] = 'image/png'
12
13     return response
```

Listing 23: Ausliefern eines geploteten Bildes mit Flask

Die Funktion wird im Blueprint argo_app der Flask-Applikation mit der Route registriert. Als Parameter `identifier` wird die eindeutige Identifikationsmnummer der Messstation erwartet. Die zum Erstellen des Graphen benötigten Daten werden aus der API angefordert. Dafür wird die URL der Datenschnittstelle der Methode `get_argo_float` in der API ermittelt. Anschließend wird unter Zuhilfenahme der Python-Bibliothek `requests` der Datensatz über einen GET-Request angefordert. Die Methode `.json()` überführt die Daten von Json in eine Python-Datenstruktur.

create_plot Diese Funktion ist für die Erstellung des Funktionsplots zuständig. Als Theme wurde sich hier für `'seaborn-whitegrid'` entschieden. Eine der damit vorgegebenen Grundeinstellungen wurden durch das Ändern der `rcParams` noch weiter angepasst.

Für die Darstellung der 3 Funktionsplots wurden 3 Achsen-Objekte erstellt. Diese Achsen werden über ein Gridspec untereinander angeordnet.

Die Datensätze werden über die Y-Achse aufgetragen, wärend die X-Achse die Zeitstempel aufzeigt. Enthält ein Datensatz nan-Werte, so wird davon ausgegangen, dass dieser Datensatz nicht vorhanden ist und diese Achse nicht gezeichnet. Überschreitet ein Datensatz den ihm zugeschriebenen Wertebereich, so wird der Wertebereich der Y-Achse fest eingestellt. Die hierfür benötigten Daten werden über die zentrale Flask-Instanz aus der Konfiguration angefordert.

Um den Platz besser nutzen zu können, wird die Beschriftung der Y-Achse nur auf dem untersten Graphen aufgetragen. Dieser enthält die Werte für den Druck des umliegenden Wassers. Dieser Messwert sollte in jeder Boje enthalten sein, es ist also nicht davon auszugehen, dass dieser Wert nicht gezeichnet wird.

Die Achsen werden über das Gridspec in einer `Figure` zusammengefasst und damit aus dem Funktionskontext zurückgegeben.

Der so erstellte Funktionsgraph wird als Canvas-Element in einen response eingebunden und über diesen ausgeliefert.

6 Testen

6.1 Funktionstest

6.2 Usability-Umfrage

7 Demonstration und Auswertung

7.1 Geschaffene Lösung

Im Rahmen dieser Arbeit wurde eine Webplattform geschaffen, die es erlaubt, die Daten des Argo-Programms durch eine Kartendarstellung der Weltmeere zu identifizieren und die Daten zur Anzeige zu bringen. Dafür wurden die Daten aus den Datensätzen extrahiert und in ein Relationales Schema überführt. Die Webpräsenz nutzt die Datenbank um die Inhalte darzustellen. In Abbildung 11 ist ein Screenshot der hier erarbeiteten Lösung zu sehen.

...

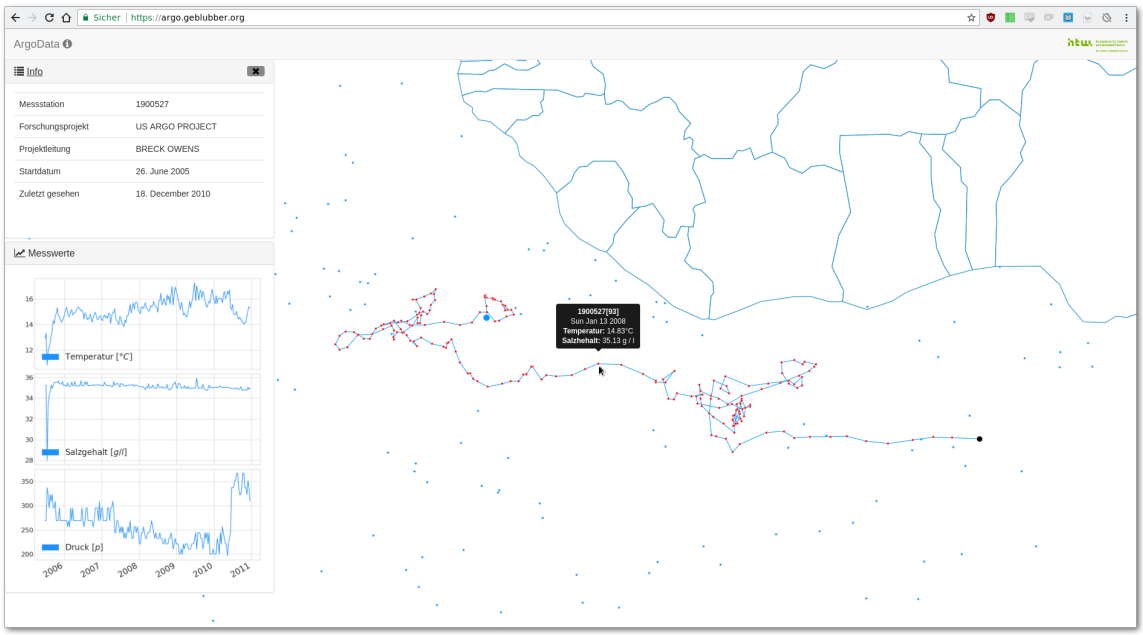

Abbildung 11: Die Webpräsenz von Argo-Data

7.1.1 Erweiterungsmöglichkeiten

7.1.2 Kritik

Der echte Zug des Wissens ist nichts Statisches, das man anhalten und in Teile zerlegen kann. Er ist immer in Fahrt. Auf einem Gleis namens Qualität. Und die Lok und die 120 Güterwagen fahren nie woanders hin, als wo das Gleis der Qualität sie hinführt.
Robert M. Pirsing – *Zen und die Kunst ein Motorrad zu warten*

8 Anhang

Quellenverzeichnis

[AGS+] Rew AuthorsRuss, Davis Glenn, Emmerson Steve, Davies Harvey, Hartnett Ed, Heimbigner Dennis, and Ward Fisher. Netcdf: Introduction and overview. https://www.unidata.ucar.edu/software/netcdf/docs/index.html.

[Arg] Argo float data and metadata from global data assembly centre (argo gdac). http://www.seanoe.org/data/00311/42182/. (Accessed on 10/09/2017).

[Arg17a] Documentation - Argo Data Management. http://www.argodatamgt.org/Documentation, Dec 2017. [Online; accessed 12. Dec. 2017].

[Arg17b] Argo data selection - Argo Data Management. http://www.argodatamgt.org/Access-to-data/Argo-data-selection, Dec 2017. [Online; accessed 12. Dec. 2017].

[CKT+15] Thierry Carval, Robert Keeley, Yasushi Takatsuki, Takashi Yoshida, Claudia Schmid, Roger Goldsmith, Annie Wong, Ann Thresher, Anh Tran, Stephen Loch, and Rebecca Mccreadie. Argo user manual, 2015.

[IBNW09] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.

[Moo98] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[net] Python cdf4. http://unidata.github.io/netcdf4-python/.

[Ng14] Jackie Ng. bootstrap-viewer-template. https://github.com/jumpinjackie/bootstrap-viewer-template, 2014.

[Ron11] Armin Ronacher. Opening the flask. http://mitsuhiko.pocoo.org/flask-pycon-2011.pdf, 2011.

[SG] Megan Scanderbeg and John Gould. A beginners' guide to accessing argo data. http://www.argo.ucsd.edu/Argo_data_guide.pdf.

[Sun16] Johan Sundstroem. world.geo.json. https://github.com/johan/world.geo.json, 2016.

[WS11] B. Witte and P. Sparla. *Vermessungskunde und Grundlagen der Statistik für das Bauwesen*. Wichmann, 2011.