Untersuchung von Interaktiven Analyse- und Visualisierungsumgebungen im Browser für NetCDF-Daten

— Masterarbeit —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Sebastian Rothe
E-Mail-Adresse:	0rothe@informatik.uni-hamburg.de
Matrikelnummer:	6223269
Studiengang:	M.Sc. Informatik
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Julian Kunkel
Betreuer:	Dr. Julian Kunkel

Hamburg, den 21.07.2016

Kurzfassung

Simulations- und Messergebnisse von Klimamodellen umfassen heutzutage oftmals große Datenmengen, die beispielsweise in `NetCDF`-Dateien als spezielle Datenstrukturen abgelegt werden können. Die Analyse dieser Messergebnisse benötigt meist komplexe und leistungsstarke Systeme, die es dem Nutzer ermöglichen, die Datenmenge an Simulationsergebnissen beispielsweise in tabellarischer Form oder durch grafische Repräsentation anschaulich darzustellen.

Moderne Cloud-Systeme bieten dem Nutzer die Möglichkeit, Ergebnisse zu speichern und beispielsweise über das Internet weltweit verfügbar zu machen. Dieses Verfahren hat allerdings auch den Nachteil, dass dazu erst die gesamte Ergebnisdatei aus dem Cloud-System angefordert werden muss, bevor sie analysiert werden kann.

Diese Arbeit befasst sich mit der Untersuchung eines alternativen Ansatzes, bei dem es für den Nutzer möglich sein soll, über eine Webanwendung erste Analysen auf serverseitig ausgeführten Werkzeugen durchzuführen, deren Ergebnisse dann im Webbrowser veranschaulicht werden können. Basis dieser `ReDaVis` (*Remote Data Visualizer*) genannten Anwendung bilden die Softwaresysteme *OpenCPU* und *h5serv*. Die Voranalysen arbeiten auf kleinen Teilmengen der Daten. Sie sollen Aufschluss darüber geben, ob detailliertere Analysen auf dem Gesamtdatensatz lohnenswert sind. Es soll untersucht werden, inwiefern vorhandene Tools diesen Ansatz bereits umsetzen können. Einige dieser Komponenten werden dann verwendet und durch eigene Komponenten ergänzt, um einen Software-Prototyp des vorgestellten Ansatzes entwickeln zu können. Dazu werden zunächst theoretische Grundlagen genauer erläutert, die dann dazu verwendet werden, die eingesetzten Komponenten als Webanwendung zusammenfassen zu können. Die Anwendung unterstützt neben Visualisierungstechniken zur grafischen Repräsentation der Datensätze auch die Möglichkeit, verschiedene aufeinanderfolgende Funktionen in Form einer Pipeline auf einen Datensatz anzuwenden.

Es wird gezeigt, inwiefern die unterschiedlichen Konstellationen an Komponenten zusammenarbeiten können oder durch Einschränkungen auf Software- und Hardwareebene ungeeignet sind beziehungsweise mit Blick auf heute weit verbreitete Alternativen nicht leistungsfähig genug arbeiten.

Danksagung

Mein besonderer Dank geht an meinen Betreuer Dr. Julian Kunkel für die Bereitstellung des Themas sowie die zahlreichen Hilfestellungen, Anregungen und Erläuterungen. Ich danke Stephan Kindermann, Carsten Ehbrecht und Andrej Fast für die unterstützenden Hinweise zu meiner Arbeit sowie die Einblicke und Erläuterungen zum Prozessierungs- und Visualisierungssystem des Deutschen Klimarechenzentrums. Außerdem möchte ich meiner Familie für die Unterstützung bei der Korrektur danken.

Inhaltsverzeichnis

1. Einleitung

Das Kapitel gibt einen Überblick über die Arbeit. In Abschnitt 1.1 wird zunächst die Relevanz des Themas erläutert. Darauffolgend definiert Abschnitt 1.2 die Ziele der Arbeit, bevor abschließend in Abschnitt 1.3 kurz der Aufbau der Arbeit dargestellt wird.

1.1. Motivation

Mit stetig wachsendem Datenaufkommen wurde es in den letzten Jahren in der Informatik immer wichtiger, leistungsfähige Methoden und Werkzeuge zur Analyse großer Datenmengen zu entwickeln. Leistungsfähige Anwendungen bieten heutzutage dem Nutzer die Möglichkeit, Datensätze mit einem großen Umfang an Funktionen zu analysieren.

Daneben werden Mess- und Simulationsergebnisse auch in Cloud-Systemen abgelegt, um sie langfristig zu speichern und beispielsweise über Webanwendungen im Internet weltweit verfügbar zu machen. Bei der traditionellen Herangehensweise wird dazu zunächst eine Auswahl der benötigten Daten getroffen, die meistens große Datenmengen umfassen, bevor die Daten heruntergeladen werden können. Die heruntergeladenen Daten lassen sich dann durch umfangreiche Analysen lokal untersuchen. Dafür müssen jedoch zwei Kriterien erfüllt sein:

1. Der in entfernten Netzwerken gespeicherte Datenbestand muss als Ganzes in Form von Dateien angefordert werden können und

2. Es muss sowohl leistungsfähige Software als auch Hardware lokal verfügbar sein, um die Analysen überhaupt durchführen zu können.

Vor allem die Soft- und Hardware ist in vielen Unternehmen nicht vorhanden. Viele Rechenzentren, die die Speicherung von Messergebnissen anbieten, verfügen jedoch oftmals sowohl über die entsprechende Hardware in Form von Clustern als auch die entsprechende Software. Gängige Werkzeuge in der Klimaforschung sind dabei *OPeNDAP* zum Anbieten der Daten oder *CDO* [Sch16] zur eigentlichen Analyse. Die Verwendung solcher Softwarelösungen haben allerdings auch Nachteile. Gerade *CDO* ist als Konsolenanwendung nur umständlich zu bedienen. Die Einarbeitung in den beachtlichen Funktionsumfang von *CDO* ist gerade für Laien sehr kompliziert.

Sind derartige Systeme auch lokal beim Nutzer vorhanden, sodass er die Analysen selbst durchführen könnte, besteht das nächste Problem darin, den benötigten Datenbestand anzufordern. Das kann bei entsprechend großen Dateien eine große Zeitspanne in Anspruch

nehmen, ohne dass der Nutzer weiß, ob die angeforderte Datei überhaupt aussagekräftige Daten beinhaltet.

Das in dieser Arbeit vorgestellte Konzept sieht vor, mit Hilfe von Voranalysen in kleinerem Umfang für diese Problematik Abhilfe schaffen. Dazu soll über Interfaces wie `REST APIs` der Zugang zu den serverseitigen, leistungsstarken Komponenten bereitgestellt werden, um die Qualität der Daten zu untersuchen. Die Anbindung der `REST APIs` über `HTTP` ermöglicht es, für die Benutzer einfach zu bedienende Web-Anwendungen bereitzustellen. Dadurch wäre es für den Nutzer möglich, mithilfe eines „einfachen“ Webbrowsers Voranalysen von Messdaten auf externen Systemen durchzuführen, bevor er sich entscheidet, welche Daten er für spätere, komplexere Untersuchungen anfordern sollte.

1.2. Zielsetzung

Ziel dieser Arbeit ist es, Ausschnitte von Klima-Daten für einen Nutzer visuell und interaktiv zugänglich im Webbrowser darzustellen. Durch gezielte Voranalysen kann dann bereits im Voraus vom Benutzer entschieden werden, ob komplexere Folgeanalysen und dementsprechend das Anfordern der gesamten Messdaten lohnenswert sind. Hierfür soll ein Werkzeug entworfen und als Prototyp entwickelt werden, welches Daten über einen bereits existierenden Service aus einer Cloud anfordert, diese Daten aufbereitet und dann dem Nutzer anschaulich ausgibt. Dabei soll untersucht werden, inwiefern Zugriffe auf große Datenmengen über den Browser in Echtzeit realisierbar sind.

Hierfür werden neuere Software-Konzepte wie *OpenCPU* und *h5serv* untersucht und weitere Komponenten bei Bedarf zunächst in vereinfachter Form entwickelt, um anschließend in verschiedenen Konstellationen zu testen, wie die Komponenten am besten zusammenarbeiten, um dem Nutzer einen möglichst effizienten Zugriff auf die Daten zu gewährleisten.

Dabei werden folgende funktionale und nicht-funktionale Anforderungen gestellt:

Funktional

- Visualisierung von Daten
  Es soll möglich sein, dem Nutzer Daten auf unterschiedliche Arten visuell präsentieren zu können. Die Daten sollen individuell auswählbar sein. Die Komponenten hinter der visuellen Darstellung sollen eine aussagekräftige Menge an Datenpunkten verarbeiten können.

- Anwendung von ausgewählten Operationen auf Daten
  Die Analyseanwendung soll dem Nutzer die Anwendung von Operationen auf angeforderte Rohdaten ermöglichen. Der Funktionsumfang soll sowohl simple Methoden als auch komplexere Funktionalitäten umfassen.

- Der Download von Daten soll ermöglicht werden
  Nach der abschließenden Voranalyse über die Webanwendung kann der Nutzer bei Bedarf die angeforderten Teildaten oder auch den gesamten Datensatz über die Anwendung herunterladen.
- Komplexe Daten sollen leicht an Dritte weiter gegeben werden
  Der Austausch von Daten und Analysen unter mehreren Nutzern kann über die Webanwendung erfolgen. Über URLs könnten außerdem externe Zugriffe auf Ergebnisse in Form von Visualisierungen realisiert werden.

Nicht-Funktional

- Daten-Sparsamkeit
  Mit dem Webbrowser des Nutzers als technisch gesehen "schwächste" Komponente muss darauf geachtet werden, dass dieser Teil der Anwendung nicht mit größeren Datenmengen oder sogar der gesamten Datenmenge in Berührung kommt. Die Verwendung von sehr kleinen Teilmengen des Datenbestands als verwertbares Ergebnis für den Webbrowser ist unumgänglich.
- Einfache Benutzbarkeit im Browser
  Die Bedienung der Benutzeroberfläche soll interaktiv sein, aber trotzdem einfach gehalten werden. Moderne Frameworks und Bibliotheken sollen den Aufbau der dynamischen Oberfläche unterstützen.
- Simplizität bei der Entwicklung der Anwendung
  Durch die genutzten Frameworks und Bibliotheken soll der Entwickler mit überschaubarem Aufwand die Entwicklung der Anwendung durchführen können. Ein modularer Aufbau der Komponenten fördert die spätere Weiterentwicklung bestehender und Neuentwicklung zusätzlicher Funktionalität.

1.3. Aufbau der Arbeit

In Kapitel 2 werden einige theoretische Grundlagen erläutert. Neben NetCDF und dem dazugehörigen Dateiformat, welches die Datenstrukturen der Mess- beziehungsweise Simulationsergebnisse speichert, werden auch die wichtigsten Konzepte und Eigenschaften des verwendeten Frameworks *AngularJS* vorgestellt.

Kapitel 3 befasst sich mit der Auswahl der zu verwendenden Komponenten für `ReDaVis` sowie mit dem strukturellen Aufbau der Anwendung, beschreibt die genutzten Komponenten und Datenstrukturen und erläutert die Funktionsweise der Anwendung.

Kapitel 4 beschreibt einige ausgewählte Aspekte der Implementation.

Kapitel 5 untersucht die Leistungsfähigkeit der Anwendung auf unterschiedlichen Ebenen und zeigt Möglichkeiten sowie Grenzen auf.

Kapitel 6 fasst die Ergebnisse der Arbeit noch einmal zusammen und gibt außerdem einen Ausblick auf Erweiterungsmöglichkeiten der Anwendung.

2. Hintergrund und verwandte Arbeiten

In diesem Kapitel werden wichtige theoretische Grundlagen wie beispielsweise NetCDF (Abschnitt 2.1.1) und AngularJS (Abschnitt 2.1.2) erläutert. Außerdem wird in Abschnitt 2.2 eine Auswahl an verwandten Arbeiten beschrieben.

2.1. Hintergrund

2.1.1. NetCDF

Das *Network Common Data Form*, kurz **NetCDF** genannt, besteht aus Softwarebibliotheken und verschiedenen Datenformaten, die es ermöglichen, wissenschaftliche Daten zu speichern und auszutauschen. NetCDF wird als offener Standard von der *Unidata Community* entwickelt und von der *University Corporation for Atmospheric Research* (UCAR) betreut.

NetCDF besteht aus:

- einem selbstbeschreibenden Dateiformat und
- Softwarebibliotheken, die die Zugriffe auf die NetCDF-Dateien ermöglichen.

Die Softwarebibliotheken werden für zahlreiche Programmiersprachen entwickelt und gewartet. So ermöglichen sie Zugriffe auf NetCDF-Dateien mit `C`, `Java` und `Fortran`, in kleinerem Umfang werden aber auch Schnittstellen für `Python`, `R`, `C++`, `Perl`, `Ruby` oder `Matlab` angeboten. Sie können unter allen gängigen Linux-, Windows- und OSX-Versionen eingesetzt werden.

Das Dateiformat speichert eine beliebige Anzahl von Daten in n-dimensionalen Arrays (mit $n = 0, 1, 2, ...$ und $n = 0$ als Skalar), um schnellen Zugriff zu ermöglichen. NetCDF ist selbstbeschreibend. Das bedeutet, dass in einer Datei über Metadaten die eigentlichen Nutzdaten beschrieben werden. Zu diesen Metadaten zählt beispielsweise die Dimensionalität der Nutzdaten. Es werden außerdem parallele Zugriffe auf NetCDF-Dateien, bestehend aus einem schreibenden und mehreren lesenden Prozessen, ermöglicht. Unidata nennt insgesamt vier NetCDF-Formate [RDE+16], die im folgenden kurz vorgestellt werden:

- Das **classic format**,

- Das **64-bit offset format**,
- Das **netCDF-4 format** und
- Das **netCDF-4 classic model format**.

Das *classic format*, welches zwischen 1989 und 2004 entwickelt wurde, ist auch heute noch das Standardformat für neue NetCDF-Dateien. Es wurde 2004 um die *64-bit offset format*-Variante erweitert, die es ermöglichte, größere Daten abzulegen und auszulesen. Das *netCDF-4 format*, welches 2008 hinzugefügt wurde, versprach eine Reihe weiterer Features, zu denen *per-variable compression*, komplexere Datentypen und eine allgemein bessere Performance zählten [RDE+16]. Ermöglicht wurde dies dadurch, dass NetCDF4 als Basis das Datenformat `HDF 5` (*Hierarchical Data Format 5*) [The16] verwendet. Dieses wird in Abschnitt 2.2.1 erläutert. Ergänzend wurde zur gleichen Zeit das *netCDF-4 classic model format* hinzugefügt, welches nur einige der Verbesserungen aus NetCDF 4 anbot und dadurch eine bessere Performanz erzielen konnte. Das Format einer NetCDF-Datei kann man sich über den `ncdump`-Befehl aus Listing 2.1 ausgeben lassen. Alternativ lässt sich das Format auch über die Interpretation der ersten vier Bytes einer NetCDF-Datei ermitteln.

```
1 ~$ ncdump -k atls14-CyGl1B.nc4
2 netCDF-4
```

Listing 2.1: Auslesen des NetCDF-Formats

Listing 2.2 zeigt einen Ausschnitt der verfügbaren Metadaten zu einer NetCDF-Datei. Die Metadaten beschreiben unter anderem die Dimensionen der Nutzdaten und welche Variablen diese Dimensionen umfassen. Neben diesen für die ganze Datei relevanten Daten sind auch Metadaten zu den einzelnen *Datasets* gespeichert. Diese können nachher verwendet werden, um die verfügbaren Rohdaten in für die angegebene Einheit sinnvolle Daten zu formatieren.

```
1  netcdf atls14-CyGl1B {
2  dimensions:
3          longitude = 480 ;
4          latitude = 241 ;
5          time = UNLIMITED ; // (1096 currently)
6  variables:
7          float longitude(longitude) ;
8                  longitude:units = "degrees_east" ;
9                  longitude:long_name = "longitude" ;
10         float latitude(latitude) ;
11                 latitude:units = "degrees_north" ;
12                 latitude:long_name = "latitude" ;
13         int time(time) ;
14                 time:units = "hours since 1900-01-01 00:00:0.0" ;
15                 time:long_name = "time" ;
16                 time:calendar = "gregorian" ;
17         short sf(time, latitude, longitude) ;
18                 sf:scale_factor = 7.3764124573405e-07 ;
19                 sf:add_offset = 0.0241695530510217 ;
20                 sf:_FillValue = -32767s ;
21                 sf:missing_value = -32767s ;
22                 sf:units = "m of water equivalent" ;
23                 sf:long_name = "Snowfall" ;
24                 sf:standard_name =
                      ↪ "lwe_thickness_of_snowfall_amount" ;
25         ...
26 }
```

Listing 2.2: Ausschnitt einer Beispielsausgabe von `ncdump -h`

2.1.2. AngularJS

Das Open-Source-Framework *AngularJS* wurde von *Google Inc.* entwickelt und erlaubt es dem Entwickler, mithilfe von `HTML` und `JavaScript` (oder `TypeScript`) eine sogenannte *Single-page-Webanwendung* zu erstellen. *Single-page* bedeutet, dass bei URL-Aufrufen diese intern bestimmten Komponenten zugeordnet werden, die dann dynamisch geladen werden. Im Gegensatz zur "herkömmlichen" Webentwicklung, bei der die Seitennavigation per URL-Aufruf jedes Mal eine neue Seite vom Server anfordert, werden unter *AngularJS* in einem Seitengerüst also nur bestimmte Templates für die Benutzeroberfläche und Komponenten für die Funktionslogik ausgetauscht, die gegebenenfalls noch mit Daten gefüllt werden, die zuvor vom Server angefordert wurden.

Das Framework verwendet das sogenannte *Model-View-ViewModel*-Muster (MVVM). Es ist eine Variante des bekannteren *Model-View-Controller*-Musters (MVC) und dient wie auch das MVC-Muster der Trennung von Programmlogik und Darstellung der Benutzer-

oberfläche. Der Unterschied ist jedoch, dass beim MVVM-Muster eine Datenbindung zwischen der *View* und dem *ViewModel* (also dem *Controller* aus MVC) besteht. Die Umsetzung des Musters sieht in *AngularJS* wie folgt aus:

- Die **Model**-Schicht beschreibt die Anwendungsdaten. Diese liegen in den meisten Fällen auf einem Server in einer Datenbank oder (wie in diesem Fall) in Dateien. Sie werden über eine `REST API` bereitgestellt.
- Die **ViewModel**-Schicht ist dafür zuständig, vom Backend oder der Datenquelle nur die Daten anzufordern, die gerade für die Benutzeroberfläche (also die *View*) benötigt werden. Tarasiewicz und Böhm ([TB14], S. 22) beschreiben die Aufgaben dieser Schicht wie folgt:
  1. *Bereitstellung und ggf. Transformation* eines Datenausschnitts der Anwendungsdaten, die auf dem Server liegen.
  2. *Bereitstellung der Funktionalität*, die im Kontext einer Anzeige benötigt wird.
- Die **View**-Schicht umfasst bei Webanwendungen im klassischen Sinn die HTML-Antwort des Servers, die vom Webbrowser des Nutzers angezeigt wird. Im Fall von *AngularJS* werden jedoch nur die angesprochenen `HTML`-Templates benötigt, die dann mit Anwendungsdaten gefüllt werden.

Die wichtigsten Komponenten in AngularJS sind für die *View* die HTML-Templates sowie für die *ViewModel*-Schicht die *Controller* und die *Services*. Sie werden daher im Folgenden kurz erläutert.

Die **Controller** bilden eine Art Zwischenschicht zwischen der *View* und dem *Model*. Zwischen ihnen und den HTML-Templates besteht auch die bereits angesprochene *Zwei-Wege-Datenbindung* (engl. *Two-Way Data Binding*). Das bedeutet, dass einerseits Änderungen im Controller für den Benutzer auf der Oberfläche sichtbar werden, andererseits aber auch Änderungen an der Oberfläche (zum Beispiel durch Benutzereingaben) direkt im Controller übernommen werden.

Ein einfaches Beispiel dazu wird in Listing 2.3 für den Aufbau des HTML-Templates sowie in Listing 2.4 für den Controller auf JavaScript-Seite gezeigt. Im Controller steht das sogenannte `$scope`-Objekt mit der Eigenschaft `name` zur Verfügung (`$scope.name`), welches per `ng-model`-Attribut des Eingabefelds angebunden und dann verändert beziehungsweise per `{{name}}` ausgelesen werden kann. Durch die Belegung von `$scope.name` im Controller in Listing 2.4 sind aufgrund der Zwei-Wege-Datenbindung bereits Eingabefeld und der Aufruf als Text mit dem Wert `"Unknown User"` vorbelegt.

Die Controller orientieren sich bezüglich des Funktionsumfangs an ihrer *View*, also dem dazugehörigen HTML-Template. Typische Aufgaben sind zum Beispiel das Anzeigen von Elementen oder Dialogen sowie erste, einfache Verarbeitung von eingegebenen oder

auszugebenden Daten. Sie sollten möglichst wenig Geschäftslogik enthalten.

Die **Services** beinhalten einen Großteil der Geschäftslogik und bieten anwendungsweit Funktionalität an. Dazu können sie von Controllern und anderen Services eingebunden werden. Ein Service stellt beispielsweise `HTTP-Request`-Funktionalität zur Verfügung, um Daten vom Backend anfordern zu können.

Das Zusammenspiel der verschiedenen Komponenten wird in Abbildung 2.1 veranschaulicht.

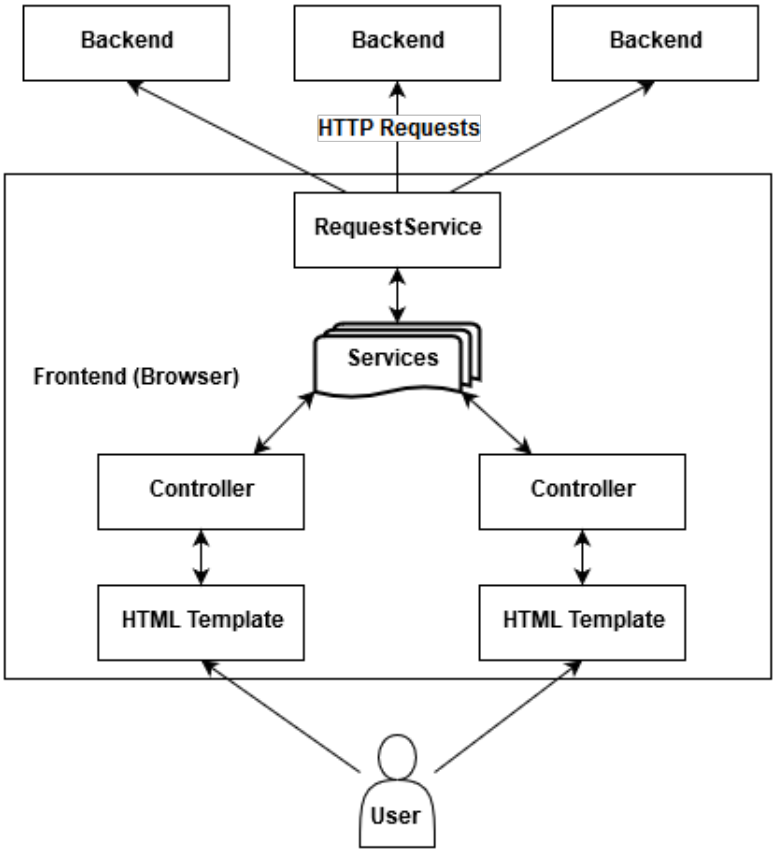

Abbildung 2.1.: Zusammenspiel der Komponenten von *AngularJS*

```
1 <html ng-app>
2     <head>...</head>
3     <body>
4         <input type="text" ng-model="name">
5
6         Hello {{name}}!
7     </body>
8 </html>
```

Listing 2.3: Einfaches Beispiel zur Zwei-Wege-Datenbindung (HTML-Template)

```
1 var msApp = angular.module('msApp', []);
2
3 msApp.controller('MainCtrl', ['$scope', function($scope){
4     $scope.name =  'Unknown User';
5 }]);
```

Listing 2.4: Einfaches Beispiel zur Zwei-Wege-Datenbindung (Controller)

2.2. Verwandte Arbeiten

2.2.1. HDF

Das **Hierarchical Data Format** (kurz HDF genannt) bezeichnet ein Datenformat, das von der *HDF group* entwickelt wird. Es wird vor allem bei der Speicherungen von großen Mengen wissenschaftlicher Daten verwendet. Heutzutage kommt das Format `HDF5`, selten auch das veraltete Format `HDF4`, zum Einsatz. *HDF* umfasst neben den eigentlichen Daten auch Metadaten, ist damit also ein selbstbeschreibendes Dateiformat.

HDF umfasst neben dem eigentlichen Dateiformat auch Werkzeuge, um die Daten verwalten, anzeigen und analysieren zu können. Dazu gehören unter anderem *h5dump*, *h5serv*, *h5cc*, *h5c++*, *h5fc* oder *HDFView*. Eins dieser Werkzeuge, *h5serv*, wird in Abschnitt 2.2.3 detailliert vorgestellt. Es sind außerdem unterschiedliche Softwarebibliotheken für *HDF* entwickelt worden, darunter für `C`, `C++`, `Java` oder `Fortran`.

HDF wurde unter anderem unter Berücksichtigung folgender Kriterien entwickelt [gro16]:

- Aufnahme großer Datenmengen und/oder komplexer Datenstrukturen
- Portabilität, um auf vielen Systemen einsetzbar zu sein
- Effiziente I/O

Eine *HDF*-Datei umfasst unterschiedliche Objekte. Die wichtigsten Objekte in *HDF* sind die *Groups* und die *Datasets*, daneben gibt es auch noch *Datatypes*, *Attributes* und *Properties*. Eine *Group* kann unterschiedliche HDF-Objekte umfassen und diese mit Metadaten genauer beschreiben. *Datasets* umfassen die Nutzdaten der Datei, die in multidimensionalen Arrays gespeichert werden. Daneben werden auch für *Datasets* Metadaten gespeichert. Der Begriff *Dataset* bezieht sich in dieser Arbeit auf ein *HDF*-Dataset und sollte daher nicht wörtlich ins Deutsche übersetzt werden.

2.2.2. OpenCPU

OpenCPU (oft mit *OCPU* abgekürzt) stellt die genutzte Backend-Komponente der entwickelten Webanwendung dieser Arbeit dar und wird seit 2009 von Jeroen Ooms entwickelt. Der *OpenCPU*-Server bietet eine `HTTP`-Schnittstelle, über die Skripte der Programmiersprache `R` ausgeführt werden können [Oom14]. `R` wird oftmals für *Data Analysis*

eingesetzt. Neben einer großen Anzahl vorinstallierter `R`-Bibliotheken hat der Nutzer die Möglichkeit, eigene Bibliotheken zu entwerfen und diese zusätzlichen Funktionalitäten über *OpenCPU* anzusprechen.

Abbildung 2.2 beschreibt die Funktionsweise von Anfragen an *OpenCPU*. Ein wichtiges Konzept, welches von *OpenCPU* aufgegriffen wird, ist das "Hinterlegen" von Ergebnissen in *Sessions*. Eine HTTP-Anfrage an *OpenCPU* resultiert daher nicht in der erwarteten Antwort. Der Server gibt dem Benutzer stattdessen einen sogenannten *session key* zurück, mit dem dann über eine zweite Anfrage das eigentliche Ergebnis der Skript-Ausführung abgerufen werden kann. Dies ermöglicht den einfachen Austausch von Ergebnissen, indem man anderen Nutzern den *session key* bereitstellt.

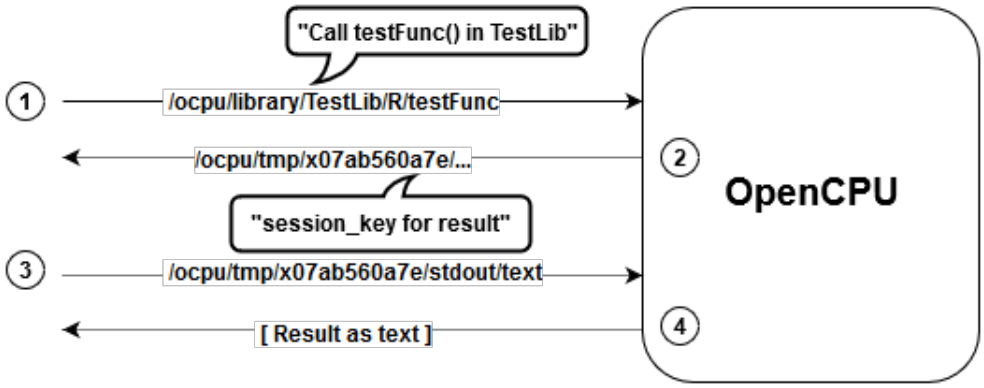

Abbildung 2.2.: Funktionsweise von Anfragen an *OpenCPU*

2.2.3. h5serv

Der *HDF REST Server*, kurz *h5serv* genannt, ist ein auf `Python` basierender Webservice, der genau wie *HDF* von der *HDF group* entwickelt wird. Er bietet ein `REST`-Interface an, mit dem `HDF5`-Daten gelesen und geschrieben werden können. Die Analyse von NetCDF-Dateien wird dadurch ermöglicht, dass das aktuelle Format *NetCDF 4* wie in Abschnitt 2.1.1 beschrieben als Basis `HDF5` nutzt. Das Interface wird über `HTTP` angesprochen. Als Standardrepräsentation der Antworten nutzt *h5serv* `JSON`.

Der Nutzer kann dabei über ein spezielles Schema in URL-ähnlichem Format sämtliche HDF5-Dateien im `/data/`-Verzeichnis des Servers ansprechen [Heb14]. Eine Anfrage in Form einer URL würde beispielsweise `https://<host>:5000/groups/feeebb9e-16a6-11e5-994e-06fc179afd5e/attributes/attr1?host=tall.public.hdfgroup.org` lauten.

Dabei kann über den Pfad in der URL innerhalb einer Datei navigiert werden und neben den eigentlichen Werten auch Attribute, Dataset-Informationen, Gruppen oder andere

Links abgefragt werden. In der Beispielabfrage werden Informationen zu einem Attribut eines Datasets abgefragt.

Über einen angehängten *Query*-Parameter wie beispielsweise

`.../?host=tall.public.hdfgroup.org`

wird die Datei ausgewählt. Die Top-Level-Domain lautet standardmäßig `hdfgroup.org`, kann jedoch auch konfiguriert werden. Die verbleibende Sub-Domain `tall.public...` muss von rechts nach links interpretiert werden und repräsentiert die Ordnerstruktur innerhalb des `/data/`-Verzeichnisses von *h5serv*. Der Query-Parameter

`?host=tall.public.hdfgroup.org`

bezieht sich also auf die Datei `tall.h5` im Verzeichnis `/data/public/`. Die Dateiendung der Dateien, die von *h5serv* verarbeitet werden sollen, kann über die Konfigurationsdatei von *h5serv* selbst eingestellt werden, der Standardfall ist `.h5`.

2.2.4. Prozessierungs- und Visualisierungssystem am DKRZ

Am Deutschen Klimarechenzentrum (DKRZ) in Hamburg wird ein Prozessierungs- und Visualisierungssystem entwickelt, das über eine REST-Schnittstelle angesprochen werden kann. Das System nutzt dabei Schnittstellen, die auf OGC-Standards basieren. Das *Open Geospatial Consortium* (OGC) ist eine Organisation, die Standards für die Verarbeitung von Geodaten festlegt. Zum Einsatz kommen dabei vor allem der *Web Processing Service* (WPS) und der *Web Map Service* (WMS).

In dem 2007 erstmals definierten WPS-Standard wird festgelegt, inwiefern Clients auf bereitgestellte Services, die Methoden zur Geodaten-Analyse anbieten, zugreifen können. Die dafür verwendete Schnittstelle soll über drei Operationen verfügen [Ope07]:

- `GetCapabilities` stellt dem Client Metadaten über den WPS-Server zur Verfügung.
- `DescribeProcess` liefert dem Client Informationen über die ausführbaren Prozesse auf dem WPS-Server.
- `Execute` erlaubt es dem Client, einen der Prozesse auf dem Server auszuführen.

Im Bereich WPS verwendet das System die *PyWPS*-Implementierung[1], um eine einheitliche Schnittstelle anbieten zu können.

Im WMS-Standard wird das Verhalten von WMS-konformen Server definiert. Diese Server stellen dem Nutzer georeferenzierte Kartenausschnitten über das Internet zur Verfügung. Hier werden zwei Operationen erwartet[dlB06]:

- `GetCapabilites` gibt dem Client Metadaten über den WMS-Server zurück.

[1] `http://pywps.org/`

- `GetMap` gibt eine Grafik zu einem Kartenausschnitt zurück. Der Ausschnitt ist über Parameter konfigurierbar.

Im Standard sind außerdem Empfehlungen zu weiteren Operationen vermerkt, dazu gehören beispielsweise Operationen, über die der Nutzer Informationen zum dargestellten Kartenausschnitt abfragen kann. Das DKRZ verwendet in seinem Visualisierungssystem *ncWMS*[2]. Zukünftig lassen sich JavaScript-Implementationen wie *OpenLayers*[3] oder *Leaflet*[4] als Client verwenden.

2.2.5. OPeNDAP

OPeNDAP[5] steht für *Open-source Project for a Network Data Access Protocol* und ist ein Framework, das den Zugriff auf entfernt gespeicherte Dateien zuzugreifen. Es wird vom gleichnamigen Unternehmen *OPeNDAP Inc.* entwickelt. Das Framework basiert auf dem *Data Access Protocol* (`DAP`), welches dazu verwendet wird, auf Daten als *name-datatype-value*-Tupel zugreifen zu können [GPS+07].

Durch die Verwendung von `HTTP` als Transportprotokoll ist es auch möglich, über das Internet per URL auf die Daten zugreifen zu können. Dazu können die Daten direkt im Browser geladen werden, was jedoch bei größeren Datenmengen schnell zum Absturz des Webbrowser führen kann. Der Aufbau der Antworten ähnelt dem Aufbau der Antworten der verwendeten Komponente *h5serv* (Abschnitt 2.2.3), orientiert sich aber im Gegensatz zum JSON von *h5serv* eher an einer C-Syntax.

Es ist möglich, über das Anhängen der Endung `.dds` an die URL die sogenannte *Dataset Descriptor Structure* (DDS) anzufordern. Dabei handelt es sich um eine Metadaten-Struktur, die die eigentlichen Nutzdaten beschreibt. Listing 2.5 zeigt die *OPeNDAP*-Antwort zu einer DDS-Anfrage, in der die einzelnen Datasets beschrieben werden. Dabei repräsentieren die Grids den Aufbau der Nutzdaten als multidimensionales Array, während die Maps für die unabhängigen Variablen wie Zeit, Breiten- und Längengrad stehen.

[2] `http://reading-escience-centre.github.io/edal-java/`
[3] `http://openlayers.org/`
[4] `http://leafletjs.com/`
[5] `http://opendap.org/`

```
Dataset {
    Grid {
      Array:
        Float64 P[time = 28][lat = 3][lon = 4];
      Maps:
        Int32 time[time = 28];
        Float32 lat[lat = 3];
        Float32 lon[lon = 4];
    } P;
    Grid {
      Array:
        Float64 T[time = 28][lat = 3][lon = 4];
      Maps:
        Int32 time[time = 28];
        Float32 lat[lat = 3];
        Float32 lon[lon = 4];
    } T;
    Float32 lat[lat = 3];
    Float32 lon[lon = 4];
    Int32 time[time = 28];
} feb.nc;
```

Listing 2.5: OPeNDAP: Antwort einer DDS-Anfrage

Der *OPeNDAP Data Server*, auch *Hyrax* genannt, basiert auf `Java`. Er ist modular aufgebaut und kann daher in seiner Funktionalität erweitert werden. Die Funktionalität der Module lässt sich in drei Bereiche unterteilen: Dateien lesen, Verarbeitung von Daten und Antworten zusammenstellen. Die Module arbeiten größtenteils mit einem Teil des Servers, dem sogenannten *Back End Server* (BES). Diese Komponente ist dafür zuständig, die Daten zu lesen und sie in einem gültigen `DAP`-Format an weitere Komponenten, wie beispielsweise dem *OPeNDAP Lightweight Front end Server* (OLFS), weiterzugeben.

OPeNDAP deckt zwar einen Großteil der gewünschten Funktionalität für den gewählten Ansatz dieser Arbeit ab, hat jedoch auch einige Nachteile. So gibt es beispielsweise eine nur schwach ausgeprägte Integration der Datenanalyse oder Visualisierung für den Browser, vereinzelt gibt es Java-Applets, die als Plugin im Browser ausgeführt werden können. *OPeNDAP* stellt größtenteils Schnittstellen für die Datenanfrage zur Verfügung. Auch sonst sind Möglichkeiten zur Datenvorschau nur begrenzt vorhanden. Man kann sich über eine sporadische Benutzeroberfläche Daten zum Exportieren ausgeben lassen, diese Daten werden dann als Datei in einem wählbaren Dateiformat (ASCII, NetCDF 3 + 4, Binary Objects (aus DAP)) heruntergeladen. Ein API für JavaScript wurde im Rahmen eines *Google Summer of Code*-Projekts angedacht, jedoch nie implementiert[6]. Der Server bietet alternativ eine JSON-API[7] an, über die eine JavaScript-Schnittstelle angebunden werden könnte.

[6] `http://docs.opendap.org/index.php/Javascript_Data_Request_Form`
[7] `http://docs.opendap.org/index.php/Hyrax_JSON`

2.2.6. CDO: Climate Data Operators

Climate Data Operators (kurz: CDO)[8] ist ein Kommandozeilen-Programm, das eine Vielzahl an Operationen zur Verarbeitung und Analyse von Klima-Daten zur Verfügung stellt. Es wird am Max-Planck-Institut für Meteorologie entwickelt und ist in der Klimaforschung weit verbreitet. CDO unterstützt unterschiedliche Datenformate, darunter auch NetCDF in der Version 3 und 4. Es kann auf allen POSIX kompatiblen Systemen eingesetzt werden.

Die Ausführung erfolgt durch den Befehl `cdo` gefolgt von einem oder mehreren Parametern, die für die gewählte Funktion stehen. Dazu werden weitere Parameter für die Funktion hinzugefügt, außerdem wird der Name der Eingabedatei übergeben sowie ein Name für die Ausgabedatei vergeben. Der Aufruf aus Listing 2.6 beschreibt beispielsweise das Exportieren eines Datasets `t2m` aus der Datei `input.nc` nach `output.nc`.

```
1 cdo selvar,t2m input.nc output.nc
```

Listing 2.6: Beispielaufruf einer CDO-Ausführung

Es ist außerdem möglich, mehrere Operatoren als Pipeline auszuführen. Dabei werden die Ergebnisse des ersten Befehls direkt an den nächsten Operator weitergegeben. Dieses Verfahren bringt zwei Vorteile [fM12]:

1. Ohne Dateien für Zwischenergebnisse reduziert sich die Ein-/Ausgabe und
2. Die Operatoren können parallel abgearbeitet werden.

Zur Pipeline-Ausführung müssen die aneinandergereihten Operatoren einfach mit einem "-" versehen werden:

```
1 cdo monmean input.nc tmp1.nc          # Calculate monthly mean
2 cdo meravg tmp1.nc tmp2.nc            # Calculate meridional average
3 cdo zonavg tmp2.nc tmp3.nc            # Calculate zonal average
4 cdo selvar,t2m tmp3.nc output.nc      # Select dataset t2m
5
6 # Execute as pipeline
7 cdo selvar,t2m -zonavg -meravg -monmean input.nc output.nc
```

Listing 2.7: CDO: Ausführung mehrerer Einzelbefehle und als Pipeline

Erfolgt die Kompilierung mit entsprechender Konfiguration, ist es außerdem möglich Plots der Ergebnisse zu erstellen. Dafür wird die *Magics++*-Bibliothek verwendet, die auf der *MAGICS*-Software des *Europäischen Zentrums für mittelfristige Wettervorhersage* (EZMW, engl. ECMWF) basiert.

[8] `https://code.zmaw.de/projects/cdo`

Zusammenfassung

In diesem Kapitel wurden mit `NetCDF` und `AngularJS` Technologien vorgestellt, sodass auf dieser theoretischen Basis aufbauend im weiteren Verlauf die Webanwendung im Detail erläutert werden kann. Darüber hinaus wurden andere verwandte Komponenten und Technologien kurz vorgestellt.

3. Design

Diese Kapitel liefert einen Überblick über den Aufbau und die Funktionsweise der Webanwendung `ReDaVis`. Dazu wird zunächst in Abschnitt 3.1 der Aufbau der Anwendung genauer erklärt. Im Anschluss werden die verschiedenen Komponenten (Abschnitt 3.2) und Datenstrukturen (Abschnitt 3.3) vorgestellt, bevor abschließend in Abschnitt 3.4 einige Abläufe innerhalb der Anwendung erläutert werden.

3.1. Aufbau der Anwendung

In diesem Abschnitt werden einige Komponenten vorgestellt, die in der gesamten Umgebung zum Einsatz kommen oder kommen können. Dazu erläutert Abschnitt 3.1.1 einige Ideen zum grundlegenden Aufbau sowie Beispiele für mögliche Subsysteme. Die verschiedenen Konstellationen, inwiefern diese Subsysteme zusammenarbeiten können, werden in Abschnitt 3.1.3 untersucht und bewertet.

3.1.1. Grundlegender Aufbau

Für eine bessere Übersicht lassen sich die benötigten Komponenten in drei Bereiche unterteilen. Diese Bereiche umfassen:

- Das **Frontend** für die Darstellung und Verwaltung der Benutzeroberfläche,
- Das **Backend** zur Verarbeitung von Daten und
- Die **Datenquelle** zur Bereitstellung dieser Daten.

Das **Frontend** umfasst alle Komponenten, die benötigt werden, um dem Nutzer das Design und die Funktionalität im Webbrowser zugänglich zu machen. Für diese Arbeit soll dafür eine eigene Webanwendung entwickelt werden. Die Benutzeroberfläche soll einfach zu verwenden sein und sich interaktiv verhalten. Neben *HTML* und *CSS* sollte also ein *JavaScript*-Framework beziehungsweise eine *JavaScript*-Bibliothek die Basis des Frontends bilden. Beispiele hierfür sind unter anderem *AngularJS*, *jQuery*, *Ember.js* oder *Backbone.js*. Für die engere Auswahl werden in Abschnitt 3.1.2 die Erfüllung verschiedener Kriterien der jeweiligen Bibliotheken miteinander verglichen.

Das **Backend** beschreibt die Werkzeuge, die für die Verarbeitungsschritte von Daten eingesetzt werden, nachdem sie aus der Datenquelle angefordert wurden. Die Funktionen zum Ausführen auf dem Datensatz sollen beispielsweise die Datenmenge reduzieren oder

umformen. Zum Einsatz kommt hierbei vor allem *OpenCPU* [Oom13a].

Die **Datenquelle** stellt die Komponenten bereit, die benötigt werden, um Daten entweder aus einer Datenbank oder - wie in diesem Fall - aus Dateien abrufen zu können. Die verwendete Komponente ist *h5serv*[1]. Eine weitere Überlegung (Abschnitt 3.1.3) war die Anbindung von *Swift*[2], dem *OpenStack Object Store project*.

In Abbildung 3.1 wird das Zusammenspiel der unterschiedlichen Komponenten aus den drei angesprochenen Bereichen im allgemeinen Fall dargestellt. Sämtliche Kommunikation erfolgt per `HTTP`-Requests. Das Frontend kommuniziert sowohl mit dem Backend als auch mit der Datenquelle. Dabei wird nach Art der angeforderten Daten unterschieden.

So werden Anfragen, die Metadaten betreffen und/oder keine zusätzliche Datenbearbeitung des Backends benötigen, direkt an die Datenquelle geschickt. Dadurch kann zusätzliche Bearbeitungszeit, die durch die Verwendung des Backends anfallen würde, eingespart werden. Werden hingegen Nutzdaten angefordert, die eine Anpassung erfordern, erfolgen diese Anfragen zunächst vom Frontend aus an das Backend. Dieses leitet die Anfrage zum Anfordern der Rohdaten an die Datenquelle weiter und bearbeitet je nach Bedarf die erhaltene Antwort, bevor sie zurück an das Frontend geschickt werden.

Abbildung 3.1.: Allgemeine Konstellation der verwendeten Komponenten

3.1.2. Auswahl des Frontend-Frameworks

In die nähere Auswahl für das Frontend-Framework fielen *AngularJS*, *jQuery* und *Ember.js*.

Tabelle 3.1 zählt verschiedene Kriterien[3] [4] auf, die für die Entwicklung dieser Anwendung relevant waren. Aufgrund dieser Auswertung sowie persönliche Erfahrung mit den Frameworks und Bibliotheken fiel die Wahl zunächst auf *AngularJS*. Im späteren Verlauf der Entwicklung wurde im Bereich der Visualisierung zusätzlich die *jQuery*-Bibliothek mit eingebunden. Im Folgenden werden einige Aspekte der Tabelle kurz erläutert.

[1] `http://h5serv.readthedocs.io/en/latest/`
[2] `http://docs.openstack.org/`
[3] `https://www.airpair.com/js/javascript-framework-comparison`
[4] `https://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks`

Kriterium	**AngularJS**	**jQuery**	**Ember.js**
Größe	144 kB	32 kB	95 kB
Web Sockets	per Erweiterung	per Erw.	per Erw.
Drag & Drop	per Erw.	✓	✓
Einfache visuelle Effekte	✓	✓	✓
Fortg. Effekte/Animation	✓	✓	per Erw.
Mobil-Support	✓	per Erw.	✓
Community-Aktivität	sehr aktiv	sehr aktiv	aktiv
Templating	✓	✗	✓
Erlernbarkeit	mittel	leicht	leicht
Persönl. Erfahrung	wenig	wenig	keine

Tabelle 3.1.: Kriterien für die Wahl des JavaScript-Frameworks

AngularJS stellt mit einer Größe von ca. 140 kB an einzubindenden Skripten das größte der zur Auswahl stehenden Frameworks und Bibliotheken. Während man früher die Größe von Bibliotheken und Frameworks durchaus berücksichtigen musste, ist dieses Merkmal bei Werten von wenigen Hundert Kilobyte für heutige Internet-Datenübertragungsraten zu vernachlässigen. In der Vergangenheit war außerdem der Support verschiedener Webbrowser und Webbrowser-Versionen ein weiteres wichtiges Kriterium. Heutzutage unterstützen die "großen" Frameworks jedoch alle gängigen Webbrowser-Versionen.

Neben der zusätzlichen Funktionalität, die die Frameworks zur Verfügung stellen, ist ein anderer wichtiger Punkt die Einfachheit der Nutzung dieser Bibliotheken. Webanwendungen sollten nach Möglichkeit einfach zu entwickeln sein. *jQuery* ist leicht zu erlernen, da die Syntax sich nur unwesentlich von JavaScript unterscheidet. *jQuery* stellt vor allem viele zusätzliche Funktionalitäten zur Verfügung, die die Arbeit des Entwickler vereinfachen.

Im Gegensatz dazu wählt *AngularJS* als Framework einen komplett anderen Ansatz. Die Frontend-Logik, die bei herkömmlichen Websites z.B. durch *PHP* oder ähnlichen Programmier- bzw. Skriptsprachen übernommen wird, wird allein durch HTML und *AngularJS* (also JavaScript) abgewickelt. Da es ein clientseitiges Framework ist, ist das Frontend vom Backend entkoppelt. Das *AngularJS*-Frontend stellt daher nur HTML-Templates mit Platzhaltern zur Verfügung, die mit Daten, die aus dem Backend an-

gefordert werden, gefüllt werden[5]. Die Webentwicklung mit *AngularJS* weißt große Unterschiede auf, da das Framework nach dem sogenannten *Model-View-ViewModel*-Muster arbeitet. Näheres dazu wird in Abschnitt 2.1.2 erläutert.

Ergänzend zu *AngularJS* und *jQuery* wurden folgende Bibliotheken ebenfalls miteingebunden:

- *Bootstrap* [OT10], ein Framework, das zahlreiche Elemente zur Benutzereingabe zur Verfügung stellt. Beispiele sind *ButtonGroups*, Icons (sogenannte *Glyphicons*), Hinweise aller Art in Form von *Alerts* oder auch Navigationsleisten.
- *angular-ui* [TAT], um die Funktionalität von *Bootstrap* in Bezug auf *AngularJS* noch zu erweitern. Erwähnenswert sind hierbei die modalen Dialoge.
- *Moment.js* [CW11], um Zeitangaben unter JavaScript besser verarbeiten zu können, da die Basisfunktionen von JavaScript nur eingeschränkte Funktionalität anbieten.

3.1.3. Mögliche Konstellationen

Konkret können neben der in Abbildung 3.1 auch andere Konstellationen an Komponenten in Erwägung gezogen werden. Im Folgenden werden einige Möglichkeiten aufgezeigt und erläutert, inwiefern eine Umsetzbarkeit gegeben ist oder bereits ausgeschlossen werden kann.

Konstellation 1: Direkte Anfragen an h5serv

Bei dem in Abbildung 3.2 gezeigten Aufbau werden Anfragen des Frontends direkt an die Datenquelle *h5serv* gestellt. Wie in Abbildung 3.1 gezeigt wurde, ist dies vor allem dann der Fall, wenn Daten angefragt werden, die nicht umgeformt werden müssen. Dies betrifft größtenteils Metadaten, die im Frontend dazu verwendet werden, die (benötigte) Struktur der NetCDF-Datei nachzubilden.

Abbildung 3.2.: Anfragen von Daten werden direkt an *h5serv* gestellt

Konstellation 2: OpenCPU als Backend

Im Gegensatz zur ersten Konstellation wird bei dem in Abbildung 3.3 dargestellten Aufbau *OpenCPU* als Backend-Komponente zwischen Frontend und Datenquelle eingefügt.

[5]Im Gegensatz dazu wird in der Webentwicklung beispielsweise mit PHP (serverseitig) eine HTML-Seite als Antwort an den Client zurückgegeben.

Dieser Zwischenschritt ist besonders dann relevant, wenn angeforderte Daten erst noch umgeformt werden müssen. Das betrifft vor allem gelesene Daten aus den Datasets der NetCDF-Dateien, die über eine Pipeline verändert werden. Die verwendete Bibliothek in *OpenCPU* kann jedoch auch genutzt werden, um Daten aus der Datenquelle einfach nur an das Frontend weiter zu reichen.

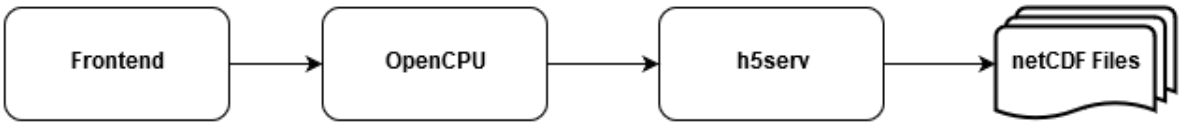

Abbildung 3.3.: Anfragen der Daten werden mithilfe von OpenCPU gestellt

Konstellation 3: Swift als Datenquelle

Abbildung 3.4 zeigt eine Konstellation, bei der als Datenquelle *h5serv* durch *Swift* ersetzt wird. Bei dieser Option muss die Anbindung der verfügbaren Daten von *Swift* per `FETCH` oder `mount` erfolgen. Diese Konstellation wurde im Folgenden nicht weiter berücksichtigt.

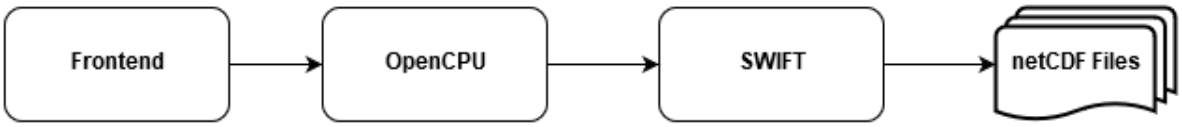

Abbildung 3.4.: Austausch von *h5serv* durch *Swift*

Konstellation 4: Vermittlung durch das Frontend

Eine weitere theoretische Anordnung zeigt Abbildung 3.5. In dieser Konstellation übernimmt das Frontend zunächst eine vermittelnde Rolle. Hierbei werden die Daten direkt von der Datenquelle angefordert, um sie dann anschließend an *OpenCPU* zu senden, wo sie verarbeitet oder analysiert werden können. Alternativ könnte man auch die angeforderten Daten zunächst im Frontend speichern und zu einem späteren Zeitpunkt weiter verwenden.

Diese Option wird jedoch in der Praxis nur für sehr kleine Datenmengen funktionieren, da der Webbrowser, also die Software, die für die Ausführung des Frontends zuständig ist, auf dem Arbeits- oder Heimrechner des Nutzers nicht in der Lage sein wird, größere Datenmengen zu verarbeiten. Um die Daten performant verarbeiten zu können, sollte die Netzwerkverbindung zwischen Backend und Frontend leistungsfähig sein[6]. Für das hier gezeigte Beispiel wäre der Webbrowser die langsamste auswählbare Komponente. Diese Konstellation wird daher nicht weiter beachtet.

[6]Im Idealfall befinden sich Backend und Datenquelle auf dem selben System.

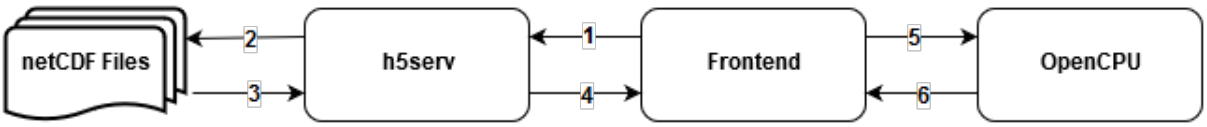

Abbildung 3.5.: Datenspeicherung im Frontend, spätere Verarbeitung in OCPU

Konstellation 5: Dateienzugriff ohne Datenquelle

Die in Abbildung 3.6 gezeigte Anordnung verzichtet auf die bereitgestellte Funktionalität der Datenquelle *h5serv*. Zugriffe auf NetCDF-Dateien erfolgen direkt durch das Backend in Form von *OpenCPU*. R-Bibliotheken wie `RNetCDF` [MW16] erweitern die Funktionalität des Backends um Zugriffe auf NetCDF-Dateien. Der zusätzliche Aufwand für die Implementation eines *h5serv*-ähnlichen Antwortschemas, mit dem das Frontend arbeiten kann, ist jedoch ein erheblicher Nachteil. Diese Option wird daher im Folgenden ebenfalls nicht weiter berücksichtigt. Sie kann allerdings in Erwägung gezogen werden, wenn der Nutzer individuelle Antwortschemen definieren möchte.

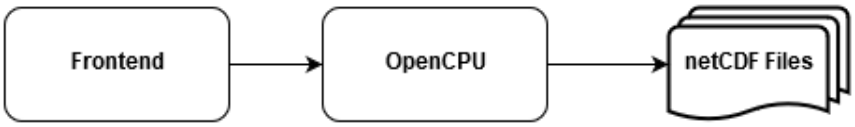

Abbildung 3.6.: Direkte Anfragenverarbeitung in OpenCPU

Konstellation 6: Import von Dateien durch Swift

Eine weitere Konstellation zeigt Abbildung 3.7. Die Idee besteht darin, weitere Dateien über SWIFT beziehen zu können. Bei dieser Konstellation kann man sich erneut die bereitgestellte Funktionalität von *h5serv* zunutze machen. Dafür müssen die weiteren NetCDF-Dateien nur über SWIFT exportiert und dann im `/data/`-Verzeichnis von *h5serv* eingefügt werden. Der restliche Ablauf innerhalb der Anwendung würde dann Konstellation 2 entsprechen.

Zu Beginn dieser Arbeit war es nicht möglich, den Datenbestand von *h5serv* im laufenden Betrieb anzupassen. Verfügbare Dateien wurden über eine Index-Datei erfasst. Diese Datei musste daher zunächst gelöscht werden, bevor beim Start von *h5serv* der Datenbestand neu erfasst wurde. Dies erforderte einen Neustart der Komponente und war somit für einen laufenden Betrieb nicht praktikabel. In einer späteren Version von *h5serv* wurde dann die Funktionalität hinzugefügt, dass neue Dateien im laufenden Betrieb erkannt werden können und die Index-Datei entsprechend angepasst wird. Änderungen am Datenbestand werden dadurch sofort sichtbar.

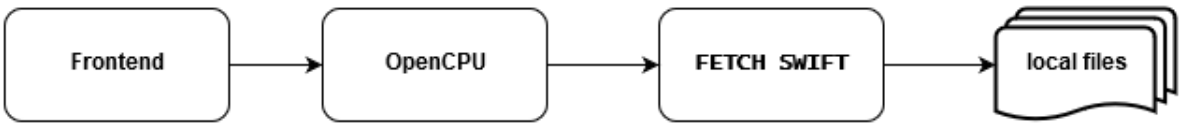

Abbildung 3.7.: Import von Dateien durch SWIFT

Konstellation 7: Zugriffsmöglichkeiten auf CDO

Ein weiterer Ansatz wird in Abbildung 3.8 gezeigt. Hierbei macht sich der Nutzer die Mächtigkeit des CDO-Werkzeugs, das in Abschnitt 2.2.6 bereits erläutert wurde, zunutze. Dafür wird in OpenCPU über die R-Bibliothek per `system call` ein CDO-Befehl ausgeführt. *h5serv* dient in diesem Zusammenhang vor allem der Repräsentation der NetCDF-Dateistruktur für die Weboberfläche. Daten für die CDO-Aufrufe werden nicht über *h5serv* angefordert. Stattdessen verweisen Dateinamen in diesen CDO-Aufrufen auf das `/data/`-Verzeichnis von *h5serv*. Die für diese Zugriffe benötigten Informationen stammen aus Metadaten, die von *h5serv* angefordert werden.

Der große Funktionsumfang von CDO würde bei entsprechend guter Anbindung dieser Aufrufe an die Weboberfläche die Entwicklung einer eigenen R-Bibliothek mit CDO-ähnlicher Funktionalität überflüssig machen. Für Testzwecke wird diese Konstellation daher in Abschnitt 5.3 genauer untersucht. Die Anbindung an die Weboberfläche erfolgte hierbei nur durch ein einfaches Eingabefeld, welches bezüglich der Dateipfade in den CDO-Aufrufen einige Anpassung vornimmt.

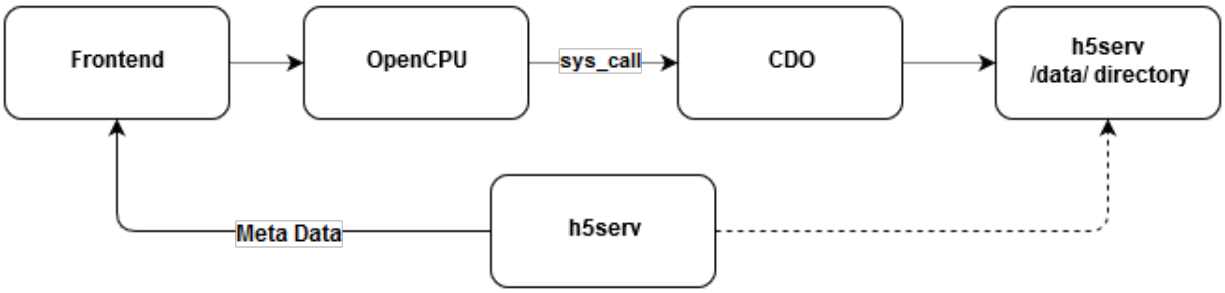

Abbildung 3.8.: Nutzung von CDO

Konstellation 8: OPeNDAP als Datenquelle

Die Verwendung von *OPeNDAP* (siehe Abschnitt 2.2.5) ist ebenfalls eine Überlegung wert. Abbildung 3.9 zeigt eine mögliche Konstellation, bei der *OPeNDAP* die Aufgabe der Datenquelle übernimmt. Anfragen können von *OpenCPU* per `REST` gestellt und Daten somit zur weiteren Analyse angefordert werden. Alternativ ist es aber wie in Abbildung 3.10 gezeigt auch möglich, dass *OPeNDAP* zusätzlich die Funktionalität des Backends, also die Datenanalyse und -anpassung, übernimmt. Damit würde dem Nutzer der gesamte

Umfang der Funktionalität von *OPeNDAP* zur Verfügung stehen. Diese Konstellation wurde jedoch für den Umfang dieser Arbeit nicht weiter verfolgt, da einerseits der Fokus auf der Verwendung von *OpenCPU* und *h5serv* lag und andererseits die Untersuchung und Nutzung eines komplexen Systems wie *OPeNDAP* sehr viel Zeit beansprucht hätte.

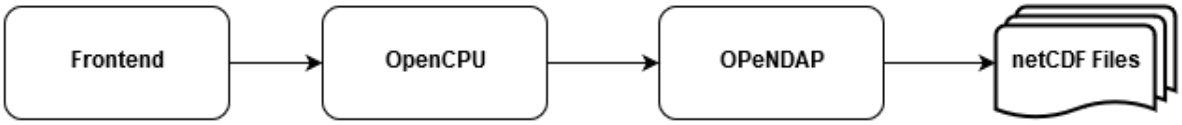

Abbildung 3.9.: Nutzung von *OPeNDAP* als Datenquelle anstelle von *h5serv*

Abbildung 3.10.: *OPeNDAP* übernimmt zusätzlich die Funktionalität des Backends

Zusammenfassung

Da der Fokus auf den beiden Komponenten *OpenCPU* und *h5serv* liegt, werden im Rahmen dieser Arbeit vor allem die Konstellationen 1 für die Metadaten-Ebene und 2 für die Datenverarbeitung beleuchtet. Ein interessanter Ansatz liefert außerdem Konstellation 7, da er den Zugriff auf den Funktionsumfang von *CDO* ermöglicht. Daher wird auch diese Konstellation untersucht.

3.2. Komponenten

3.2.1. Frontend

Der grundlegende Aufbau sowie das Zusammenspiel der verwendten Services und Controller werden in Abbildung 3.11 gezeigt. Abgebildet sind die wichtigsten Komponenten, die an der grundlegenden Verarbeitung von Daten aus den NetCDF-Dateien beteiligt sind. Daneben gibt es noch weitere, nicht abgebildete Services. Bei Komponenten mit gestricheltem Rahmen handelt es sich um externe, bereits existierende Ressourcen, wie etwa Bibliotheken für die Visualisierung oder die Backend-Anwendungen *OpenCPU* und *h5serv*.

Im Folgenden werden die einzelnen Komponenten des Frontends kurz vorgestellt, eine detaillierte Beschreibung folgt in Abschnitt 4.2. Es handelt sich um Services sowie um ausgewählte Controller. Die Services und Controller werden dabei in Kategorien unterteilt:

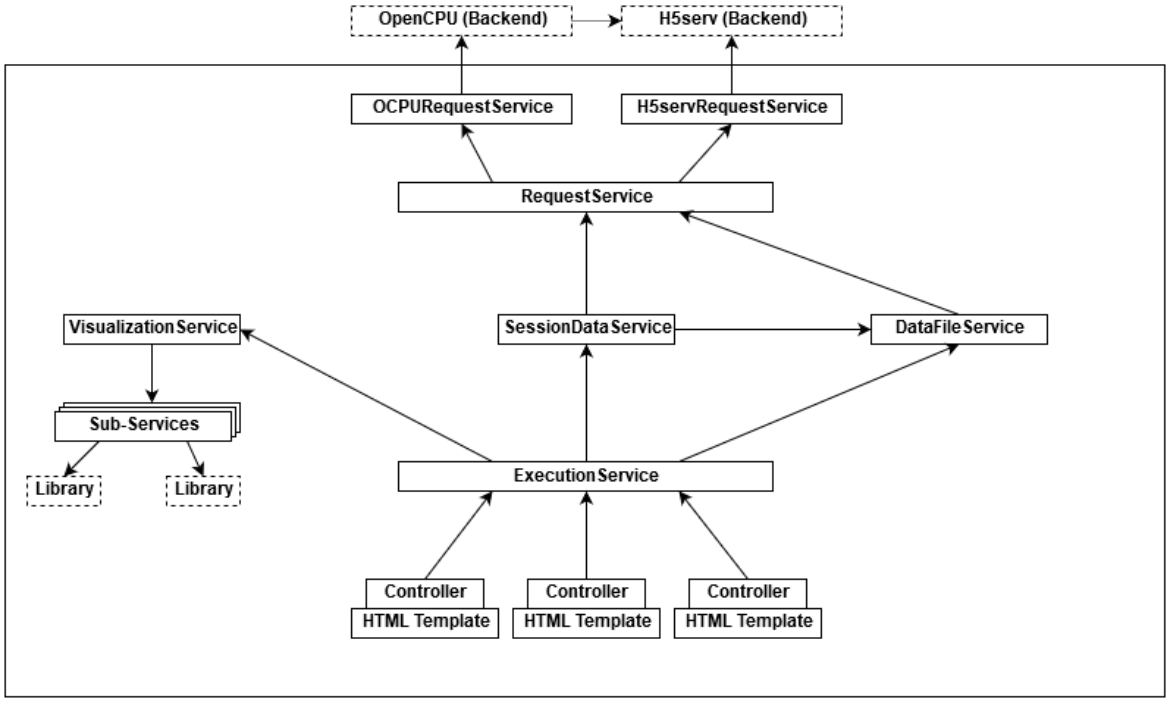

Abbildung 3.11.: Aufbau des Frontends

Steuerkomponenten

Die Steuerkomponente verwaltet die wichtigsten Komponenten des Frontends, steuert wichtige Teile der Anwendung und bietet außerdem den Controllern eine umfangreiche Anlaufstelle für einen Großteil der Verwaltungsfunktionalität. Sie umfasst nur einen Service:

- **ExecutionService**
  Zentrale Anlaufstelle für Controller; Verwaltung von Services

Verwaltungskomponenten

Die Verwaltungskomponenten verwalten einen Großteil der Metadaten, die im Frontend benötigt werden, um ein Abbild der ausgewählten Datei und Datensätze repräsentieren zu können. Diese Kategorie beinhaltet die folgenden Komponenten:

- **RequestService**
  Schnittstelle zur Kommunikation mit Backend-Komponenten
- **SessionDataService**
  Verwaltung der Informationen zur aktuell verwendeten NetCDF-Datei
- **DataFileService**
  Verwaltung der verfügbaren NetCDF-Dateien

- **DatasetController**
  Modaler Dialog zur Auswahl eines Basis-Datasets
- **PipelineFunctionController**
  Modaler Dialog zur Auswahl einer Pipeline-Funktion

Datenverarbeitungskomponenten

Die Komponenten dieser Kategorie nutzen bereitgestellte Informationen der Verwaltungskomponenten sowie Benutzereingaben, um die Datenverarbeitung selbst durchführen zu können oder Backend-Komponenten mit dieser Aufgabe zu beauftragen. Zu dieser Kategorie zählen:

- **PipelineService**
  Verwaltung der im Backend ausführbaren Pipeline
- **PreviewController**
  Bereitstellung der Funktionalität der Vorschauseite
- **VisualizationService**
  Schnittstelle zur Verwendung der Visualisierungsbibliotheken

Überwachende Komponente

Die überwachende Komponente protokolliert HTTP-Requests des Frontends an Backend-Komponenten. Diese Aufgabe wird nur von einem Service übernommen:

- **PerformanceService**
  Komponente zur Erfassung und Messung von `HTTP`-Requests

Unterstützende Komponenten

Unterstützende Komponenten stellen allgemein benötigte und wiederkehrende Funktionalität zur Verfügung und kümmern sich außerdem um Logging und Anwendungsmeldungen für den Nutzer. Die Kategorie umfasst die folgenden Komponenten:

- **LoggingService und AlertService**
  Komponenten zur Logging- und Weboberflächen-Ausgabe
- **ConfigService**
  Verwaltet Konfigurationseinträge der Webanwendung
- **SnapshotService**
  Verwaltet zwischengespeicherte Visualisierungsergebnisse
- **HelperService**
  Bereitstellung von unterstützender Funktionalität

3.2.2. Backend - OpenCPU

Für die Verarbeitung von Daten wurde als Backend die in Abschnitt 2.2.2 vorgestellte Komponente *OpenCPU* verwendet. Er wird einerseits vom Frontend über HTTP-Requests angesprochen, kann andererseits aber auch selbst Anfragen über HTTP an die Datenquelle stellen, um Daten anfordern zu können. Für diese Funktionalität wurde die `R`-Bibliothek `H5servBackend` entwickelt. Diese ermöglicht das Abfragen von Daten aus *h5serv* (einfache Weiterleitung durch *OpenCPU*) und stellt außerdem die Funktionen zur Abarbeitung einer übergebenen Pipeline zur Verfügung. Details zur Implementation werden in Abschnitt 4.3 kurz vorgestellt.

3.2.3. Datenquelle - h5serv

Als Datenquelle wird die Komponente *h5serv* genutzt, deren Funktionsweise in Abschnitt 2.2.3 erläutert wurde. *h5serv* ermöglicht Zugriffe auf sämtliche NetCDF-Dateien im `/data/`-Verzeichnis und kann daher sowohl für Anfragen im Bezug auf Metadaten verwendet werden als auch die eigentlichen Nutzdaten aus den Dateien bereitstellen. Während das Backend nur mit den Nutzdaten arbeitet, repräsentiert das Frontend auf Basis der Metadaten die Struktur der NetCDF-Datei. Mit Hilfe der Metadaten können dann auch vom Frontend aus Nutzdaten für erste Visualisierungen angefordert werden. *h5serv* nutzt JSON als Format für Antworten und kann damit sehr einfach von JavaScript, welches im Frontend eingesetzt wird, interpretiert werden.

3.3. Datenstrukturen

Datenstrukturen werden in *JavaScript* beziehungsweise *AngularJS* meistens als Objekte mit verschiedenen Attributen (und gegebenenfalls auch Funktionen) dargestellt. Da diese Objekte jederzeit änder- und erweiterbar sind, kann man sie nicht als “feste“ Datenstrukturen beschreiben. Trotzdem kommen in `ReDaVis` immer wieder Objekte vor, die im Kern die gleiche Struktur aufweisen. Im Folgenden werden davon einige wichtige vorgestellt.

3.3.1. Metadaten

```
var metadata_object =
{
  "target": "http://.../datasets/<id>?host=...",
  "title": "sund",
  "href": "...",
  "id": "cc28ad46-19ca-11e6-b44f-0025905ac4c4",
  "dataset_info": {
    "creationProperties": {
      "fillValue": -32767,
    },
    "shape": {
      "dims": [1096,241,480],
      "maxdims": [0,241,480],
    },
    "preview_link":
      ↪ "...<id>/value?select=[0:1,0:10,0:100]&host=...",
    "dimension_list": [
      ["datasets/cc450054-19ca-11e6-b44f-0025905ac4c4"],
      ["datasets/cc57bd0c-19ca-11e6-b44f-0025905ac4c4"],
      ["datasets/cc57a07e-19ca-11e6-b44f-0025905ac4c4"]
    ],
    "attributes": [
      { "name": "scale_factor", "value": 0.6592098637327759 },
      { "name": "add_offset", "value": 21599.670395068133 },
      { "name": "units", "value": "s" },
      { "name": "long_name", "value": "Sunshine duration" }
    ],
    "units": "s",
    "long_name": "Sunshine duration"
  }
}
```

Listing 3.1: Beispiel zu den Metadaten eines Datasets

Listing 3.1 zeigt einen Ausschnitt aus den im Frontend verfügbaren Metadaten zum mehrdimensionalen Dataset `sund`. Die Metadaten werden in einer dem Listing ähnlichen Form als JSON-String vom Backend als Antwort gesendet und dann dort im *SessionDataService* gespeichert. Ihr Aufbau erinnert an den Header einer NetCDF-Datei aus Listing 2.2, es sind viele Attribute wiedererkennbar. Dieser Service führt jedoch für eine solche Datenstruktur nicht nur eine Anfrage durch. Die Anfrage für die grundlegenden Informationen enthält unter dem Attribut `href` eine URL, unter der weitere URLs für die zusätzlichen Informationen zu finden sind. Der schlussendliche Zusammenbau der finalen Struktur ist daher ein komplexer Vorgang, der aus zahlreichen Anfragen besteht.

Neben essentiellen Informationen wie einem Kurznamen (hier `title` genannt) sowie

URLs, unter denen die weiteren Informationen zum Dataset abgerufen werden können, beinhaltet das Metadaten-Objekt wichtige Informationen zur Struktur der meist mehrdimensionalen Daten. Unter `metadata_object.dataset_info.shape.dims` lässt sich diese Struktur ablesen: die Daten werden in einem drei-dimensionalen Array gespeichert, das insgesamt $1096 \cdot 241 \cdot 480 = 126785280$ Werte halten kann. Die einzelnen Dimensionen werden hierbei den unabhängigen Datasets zugeordnet. In diesem Beispiel handelt es sich dabei um `time`, `long` und `lat`, also Zeitpunkten sowie Längen- und Breitengrad. In der `dimension_list` sind Verlinkungen zu diesen Datasets zu finden.

Unter der Kategorie `attributes` werden weitere für das Dataset wichtige Informationen abgelegt. Dazu gehören `scale_factor` und `add_offset`. Diese Werte werden benötigt, um die Rohdaten umrechnen zu können, damit sie für die Einheit sinnvolle Daten ergeben. Die Umrechnung sieht wie folgt aus:

$$formatted\_data = raw\_data \cdot scale\_factor + add\_offset$$

Die Einheit wird ebenfalls als Attribut unter `units` gespeichert. Auch eine ausgeschriebene Version des Dataset-Namens wird als Attribut abgelegt.

3.3.2. Basis-Datensatz

```
var base_ds = {
    request_path: ...,
    host: ...,
    dataset_title: ...,
    axes: [],
    multi_dim: ...,
    dim_val_sel: ...,
    axis_selection: ...,
    adjustValues: [],
    fillValue: ...,
    values: []
};
```

Listing 3.2: Attributbelegung eines Basis-Datensatzes

Die Basis-Datensatz-Struktur bildet die Grundlage jedes Visualisierungsvorgangs und jeder Pipeline. Sie wird über den in Abschnitt 4.2 beschriebenen *DatasetController* erstellt. Die Struktur verfügt über zahlreiche Attribute, die in verschiedensten Services benötigt werden, teilweise aber auch nur für die Anzeige auf der Benutzeroberfläche gespeichert werden. Dazu zählt beispielsweise der `dataset_title`.

`request_path` und `host` werden intern benötigt, um daraus weitere Anfragen an das Backend bezüglich dieses Datasets erstellen zu können. Dies wird vor allem zum Abruf der eigentlichen Daten benötigt.

Die Attribute `axes`, `multi_dim` und `axis_selection` halten Informationen über den

schlussendlichen Typ der Visualisierung bereit. Es wird unter anderem gespeichert, welche Datasets auf die Achsen gelegt werden und wie viele Achsen später repräsentiert werden sollen.

Ein Liniendiagramm verfügt zum Beispiel über eine Achse, die einer unabhängigen Dimension entspricht, deren Werte jeweils einem Wert des Datasets (auf der anderen Achse) zugeordnet werden. Im Gegensatz dazu können bei einer Heatmap zwei unabhängige Dimensionen auf den Achsen dargestellt werden. Für eine Karte bieten sich hierfür Längen- und Breitengrad an. Die eigentlichen Werte des Datasets werden dann über die Farbe am jeweiligen Punkt der Heatmap repräsentiert. In beiden Fällen müssen für die restlichen unabhängigen Dimensionen feste Werte gewählt werden.

Die gewählten Werte werden im Attribut `values` gespeichert. Im Beispiel `values = [-1,10,40]` mit den Dimensionen `time/lat/long` repräsentieren die Werte `10` beziehungsweise `40` die Indizes des Datasets `lat` beziehungsweise `long`. Die eigentlichen Werte hinter diesen Indizes müssen später explizit zugeordnet werden, intern benötigen diese Abfragen jedoch die Indizes und können daher nicht direkt mit den Werten arbeiten. Der Wert `-1` für die Zeit bedeutet hingegen, dass für diese Dimensionen alle Indizes relevant sind. Es handelt sich hierbei also um eine der Dimensionen, die einer Achse zugeordnet wird.

`adjustValues` und `fillValue` werden für die Datenformatierung benötigt. Das erste Attribut beschreibt, ob und wenn ja, wie die Rohdaten formatiert werden sollen (siehe `scale_factor` und `add_offset` im Abschnitt 3.3.1), das zweite Attribut speichert den Füllwert des Datasets. Dieser Wert wird automatisch vergeben, wenn kein Messwert vorhanden ist. Er muss dementsprechend bei einer Auswertung entsprechend berücksichtigt werden, damit er nicht als vermeintlicher Messwert behandelt wird.

3.3.3. Pipeline-Funktion

```
{
name: 'average_1d',
heading: 'average_1d',
description: 'Determines the average for the selected secondary
   ↪ axis over all the requested values.',
options:{
    usable: true,
    show: true,
    alt_ds: false
},
axis_sel_types: [
    cnstlst.AXIS_COMPLETE_DS
],
params:[
    { type: cnstlst.PL_PARAM_DUMMY_VALUE, id: 'dim', name:
       ↪ 'Dimension', value: -1, show: false }
]
}
```

Listing 3.3: Aufbau einer Pipeline-Funktion, Beispiel *average_1d*

Listing 3.3 zeigt den Aufbau einer Pipeline-Funktion. Neben dieser Funktion werden die anderen Funktionen, die allesamt die gleiche Attributsstruktur aufweisen, des *PipelineService* in einem Array verwaltet. Die unterschiedlichen aufgelisteten Attribute haben für die Programmlogik unterschiedliche Zwecke. Während die `options` und die `axis_sel_types` für die Funktionalität des Frontends wichtig sind, werden die Parameter aus dem Array `params` an *OpenCPU* übergeben, wo sie dann bei der Abarbeitung der Pipeline je nach Funktion entsprechend verarbeitet werden.

Die Attribute `name`, `heading` und `description` werden für die Benutzeroberfläche benötigt. Während `name` die interne Bezeichnung darstellt, handelt es sich bei `heading` um den Anzeigenamen der Funktion. `description` erläutert dem Nutzer kurz die Funktionsweise der Funktion.

Die `options` sind dafür zuständig, inwiefern die Funktion in der Benutzeroberfläche unter welchen Umständen auswählbar ist. `usable` beschreibt, ob die Funktion in der Pipeline nutzbar ist, `show` hingegen gibt an, ob der Nutzer die Funktion selbst auswählen darf. `alt_ds` ist momentan noch ohne Funktion, gibt jedoch später bei gesetztem Flag an, dass für diese Funktion ein weiteres Dataset ausgewählt werden darf. Dies wird beispielsweise für Funktionen interessant, in denen eine Operation auf zwei Datasets werteweise ausgeführt wird und man als Ergebnis ein neues Dataset erhalten würde. Beispiele hierfür wären Vergleiche wie Minimum, Maximum, aber auch kleinere Berechnungen wie die Differenz oder der Mittelwert.
Das Array `axis_sel_types` beschreibt, für welche Art von Dataset-Selektion diese

Funktion verwendbar ist. Momentan sind in der Anwendung drei verschiedene Typen definiert:

- `AXIS_DS_INDEP_DIM`:
  Das Basis-Dataset verwendet eine 2D-Visualisierung, bei der auf einer Achse eine unabhängige Dimension und auf der anderen Achse die Werte des Datasets angebracht werden.
- `AXIS_TWO_INDEP_DIMS`:
  Hierbei werden für beide Achsen unabhängige Dimensionen verwendet. Dies ist beispielsweise für eine Heatmap der Fall.
- `AXIS_COMPLETE_DS`:
  Das Basis-Dataset umfasst zunächst ein gesamtes Dataset. Für entsprechende Visualisierungsmethoden müssen diese mehrdimensionalen Daten jedoch durch entsprechende Pipeline-Funktionen auf zwei- oder dreidimensionale Daten reduziert werden.

Als Beispiel ist die oben genannte Funktion nur für gesamte Datasets anwendbar, da ihre Aufgabe darin besteht, die Dimensionen eines Datasets zu reduzieren. Eine einfache Funktion wie `addition_1d` ist hingegen für alle Typen anwendbar, da für alle selektierten Werte eine Addition durchgeführt werden kann.

Die letzte Attributgruppe umfasst die Parameterliste `params`. Diese Parameter werden bei der Ausführung der Pipeline als Funktionsparameter verwendet. Sie müssen daher vom Benutzer festgelegt werden. Es sind hierbei verschiedene Typen vorgesehen, die später in der Benutzerfläche unterschiedlich eingegeben werden. Während sich für einen nummerischen Wert beispielsweise ein Schieberegler anbietet, können `boolean`-Werte über eine *Checkbox* festgelegt werden. Wichtig ist auch, ob bei einem Parameter das `show`-Flag gesetzt wurde. Nur wenn dies der Fall ist, kann der Nutzer über die Benutzeroberfläche selbst den Wert für diesen Parameter festlegen. Für den Fall `show == false` wird der Wert intern von der Anwendung selbst bestimmt.

3.4. Ablauf

In diesem Abschnitt wird auf verschiedene Abläufe innerhalb der Anwendung eingegangen. Dabei werden zum grundsätzlichen Verständnis zunächst die Abläufe bei Anfragen an Backendkomponenten aufgegriffen, bevor darauf basierend komplexere Abläufe wie die Vorschau-Funktion oder das Abarbeiten einer Pipeline erklärt werden.

3.4.1. Ablauf einer h5serv-Anfrage

Anfragen an *h5serv* erfolgen über HTTP-Requests. Das Frontend schickt dafür zunächst in einem URL-ähnlichen Pfad sämtliche relevanten Parameter an *h5serv*. Listing 3.4 zeigt

eine einfache Beispielanfrage an *h5serv* als `cURL`-Aufruf[7].

```
1 ~$ curl -X GET http://localhost:5000
    ↪ /?host=atls14-CyGl1B.public.hdfgroup.org
```

Listing 3.4: Beispielanfrage an *h5serv* mit `curl`

Die Antwort von *h5serv* erfolgt als JSON-String und lässt sich daher von einem JavaScript-Frontend sehr gut verarbeiten. Dabei besteht die Antwort üblicherweise aus zwei Teilen. Im ersten Teil der Antwort gibt *h5serv* nützliche URLs zurück, die der Nutzer direkt als weitere Anfrage wieder absenden kann. Dadurch kann er durch weitere Teile der aktuellen NetCDF-Datei navigieren. Der zweite Teil der Antwort umfasst die relevanten Daten der aktuellen Anfrage. Dabei kann es sich je nach Anfrage um Daten aus Datasets, Informationen zur Dataset-Struktur oder auch Metadaten zu den Dateien im `/data/`-Verzeichnis von *h5serv* handeln.

Das Antwortschema von *h5serv* ermöglicht es dem Nutzer also im Allgemeinen, sich mithilfe von URLs ähnlich wie auf einer Website durch die NetCDF-Dateien zu navigieren.

3.4.2. Ablauf einer OpenCPU-Anfrage

In einem ersten Schritt werden Anfragen im Frontend ähnlich wie bei Anfragen an *h5serv* als URL formuliert. Die URL repräsentiert dabei einen bestimmten Funktionsaufruf innerhalb einer bestimmten `R`-Bibliothek. Für die Funktion wichtige Parameter werden jedoch nicht über die URL ausgelesen. Sie werden über das `data`-Feld übergeben. Die selbe Funktionsweise nutzen Browser bei POST-Requests, in denen eingegebene Daten aus Formularen angehängt werden müssen.

OpenCPU führt dann die aufgerufene R-Funktion aus. Je nach aufgerufener Funktion kann es sein, dass *OpenCPU* selbst HTTP-Requests an weitere Komponenten wie beispielsweise *h5serv* verschickt, um weitere Daten anzufordern. Als Antwort an den Nutzer sendet *OpenCPU* einen Session-Key, mit dem das eigentliche Ergebnis dann in einer zweiten Anfrage an *OpenCPU* abgerufen werden kann. Dieses Hinterlegen von Daten in Sessions soll es den Nutzern ermöglichen, die Ergebnisse untereinander austauschen zu können, indem der Session-Key weitergegeben wird. Die Ergebnisse der Sessions sind mit unveränderten Einstellungen für 24 Stunden verfügbar.

3.4.3. Ablauf einer Vorschauerzeugung

Bei einer Vorschauerzeugung werden mehrere Anfragen, wie sie zuvor bereits beschrieben wurden, an Backend-Komponenten benötigt. Dabei erfolgt ein schrittweises Erschließen sämtlicher benötigter Parameter durch die Eingaben des Benutzers. Das bedeutet, dass

[7] `cURL` (*Client for URLs*) ist ein Kommandozeilen-Programm, das in diesem Fall ähnlich wie ein Browser eingesetzt werden kann. Im Gegensatz zum Browser gibt `cURL` jedoch die Antwort eines Webservers in Textform aus.

jede Eingabe, die der Nutzer tätigt, weitere Metadaten lädt, aus denen sich der Nutzer erneut einige aussuchen muss, bevor der nächste Schritt erfolgen kann.

Der Nutzer öffnet zunächst den in Abbildung 3.12 gezeigten Dialog zur Auswahl eines Datasets. Der Dialog fragt den Nutzer zuerst nach der zu verwendenden Datei. Die Liste mit allen verfügbaren Dateien wird dafür zu Beginn vom *ExecutionService* angefordert. Nach der Auswahl einer Datei beauftragt der *ExecutionService* den *SessionDataService*, die Struktur der ausgewählten NetCDF-Datei zu erschließen.
Dies wird durch den *SessionDataService* in mehreren Schritten erreicht:

1. Ermitteln des `root link` der Datei für alle folgenden Anfragen
2. Ermitteln der verfügbaren Datasets
3. Abfrage grundlegender Informationen jedes einzelnen Datasets
4. Überprüfen auf die Verfügbarkeit einer `dimension_list` (pro Dataset)
5. Wenn verfügbar: Laden der `dimension_list` (pro Dataset)
6. Ermitteln der benötigten Attribute (pro Dataset)
7. Abfrage der Verfügbarkeit der Attribute (pro Dataset)
8. Laden der vorhandenen Attribute (pro Dataset)

Die grundlegende Dataset-Struktur der Datei ist nun vorhanden und der Nutzer kann weitere Parameter auswählen. Als nächstes wird das grundlegende Dataset ausgewählt, deren Werte analysiert werden sollen. Der Nutzer hat außerdem die Option, die Rohdaten anzupassen, damit sie für die Einheit des Datasets sinnvoll verwendbar sind. Abschließend müssen im Dialog noch die unabhängigen Dimensionen des ausgewählten Datasets verarbeitet werden. Dazu wählt der Nutzer eine Dimension, beispielsweise die Zeit, für die X-Achse aus, während für die verbleibenden Dimensionen feste Werte gewählt werden.

Sind alle Einstellungen bezüglich der zu visualisierenden Daten getroffen, müssen noch die Visualisierungsmöglichkeiten gewählt werden. Der Nutzer kann hier zunächst eine Visualisierungsbibliothek auswählen. Diese entsprechen den Sub-Services des *VisualizationServices*. Jede Bibliothek stellt verschiedene Visualisierungstypen zur Verfügung. Bei der Auswahl eines Typs werden zunächst dem *VisualizationService* die angeforderten Daten und ein Metadaten-Objekt übergeben. Der Service reicht die Daten und Informationen an den entsprechenden Sub-Service weiter, der dann die Visualisierung vornimmt. Dafür müssen die Daten zunächst in ein geeignetes Format umgewandelt werden, bevor der Sub-Service sie als Datenpunkte für die graphische Darstellung verwenden kann. Jeder Sub-Service verfügt dazu über eine eigene Funktion, die diese Umformung durchführt. Die Visualisierung wird abschließend in einem entsprechenden HTML-Element eingehängt.

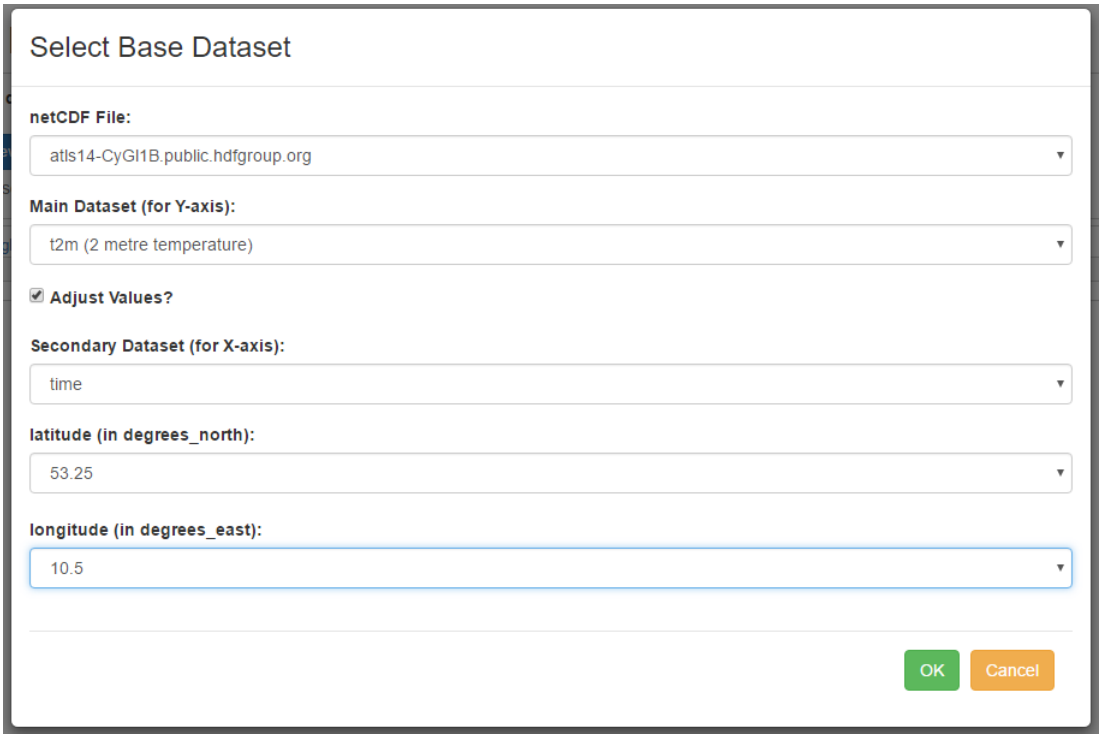

Abbildung 3.12.: Dialog zur Auswahl eines Basis-Datasets

Die Auswahl des *FlotchartVisualizationServices* als Bibliothek ermöglicht beispielsweise eine in Abbildung 3.13 gezeigte interaktive Repräsentation in Form eines Liniendiagramms. Hier wird nur ein Ausschnitt der Daten betrachtet, der zuvor interaktiv im gesamten Diagramm ausgewählt wurde. Das Bewegen der Maus über einen Datenpunkt (siehe kleiner blauer Kreis) zeigt die genauen Informationen zu diesem Punkt unterhalb des Diagramms an.

3.4.4. Ablauf einer Pipelineabarbeitung

Die Abarbeitung einer Pipeline erfolgt ähnlich wie bei der in Abschnitt 3.4.3 beschriebenen Visualisierung. Der Unterschied liegt darin, dass zwischen der Auswahl des Basis-Datasets und der Visualisierung vom Nutzer ausgewählte Funktionen auf den Rohdaten ausgeführt werden. Diese Ausführung erfolgt von *OpenCPU* im Backend, die Zusammenstellung der Pipeline über die in Abbildung 3.14 gezeigte Oberfläche.

Nachdem der Nutzer das Basis-Dataset ausgewählt hat, kann er der Pipeline zahlreiche Funktionen hinzufügen, die auf dem Datensatz ausgeführt werden sollen. Die Begrenzung an Schritten lässt sich über den *ConfigService* festlegen, der Standardwert beträgt 20 Schritte.

In der gezeigten Oberfläche kann der Nutzer eine Übersicht über alle Schritte erhalten.

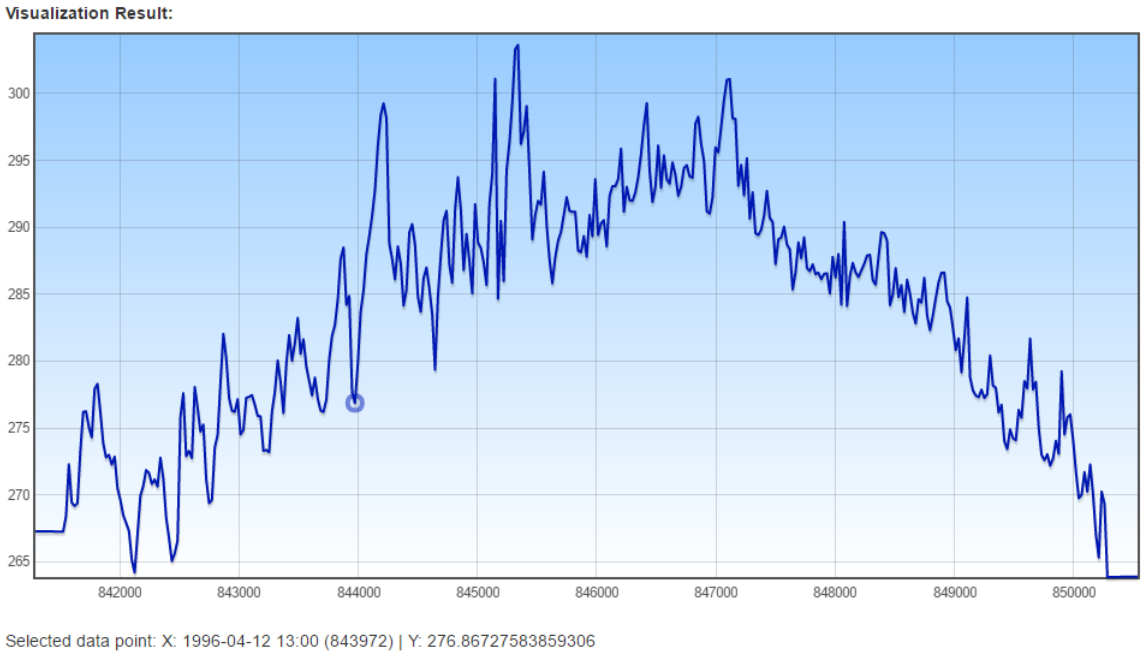

Abbildung 3.13.: Visualisierungsbeispiel: Temperatur in Kelvin für Lüneburg (1996). Die X-Achse beschreibt die Anzahl der Stunden seit dem 1. Januar 1900.

Details zu einem einzelnen Schritt lassen sich per Klick auf den Schritt einblenden (beispielsweise wie in Schritt 3 in Abbildung 3.14). Über diese Detailansicht lassen sich die Schritte anpassen, löschen oder in der Reihenfolge der Pipeline-Ausführung verschieben. Unterhalb des Pipeline-Aufbaus kann der Nutzer über die Schaltfläche *Execute Pipeline* die Ausführung der Pipeline starten. Sobald die Abarbeitung abgeschlossen ist, lässt sich das Ergebnis mit den zuvor bereits beschriebenen Abläufen aus Abschnitt 3.4.3 visualisieren.

Intern muss jedoch zunächst beim Absenden der Pipeline diese in ein JSON-ähnliches Format umgewandelt werden. Der entstandene String wird dann als Parameter dem Funktionsaufruf für *OpenCPU* (siehe Abschnitt 3.4.2) übergeben. Die `R`-Bibliothek *H5servBackend*, die von *OpenCPU* aufgerufen wird, parst den übergebenen Pipeline-JSON-String und erhält so eine `R`-gültige Datenstruktur, die die Pipeline widerspiegelt. Die einzelnen Schritte werden dann der Reihe nach abgearbeitet, dabei werden die Ergebnisse des letzten Schritts dem aktuellen übergeben.

Nach Ausführung des letzten Schritts wird das Resultat erneut in ein JSON-String formatiert und als Antwort an das Frontend zurückgesendet. Die Bibliothek *H5servBackend* verfügt über eine Überprüfung, die abschließende Antworten an das Frontend verwirft, wenn sie eine bestimmte Größe überschreiten. Dies dient dem Zweck, den Webbrowser des Nutzers nicht mit einer zu großen Antwort zum Absturz zu bringen. Der Standardwert liegt bei 10 MB. Ab dieser Größe kann es bereits passieren, dass einzelne

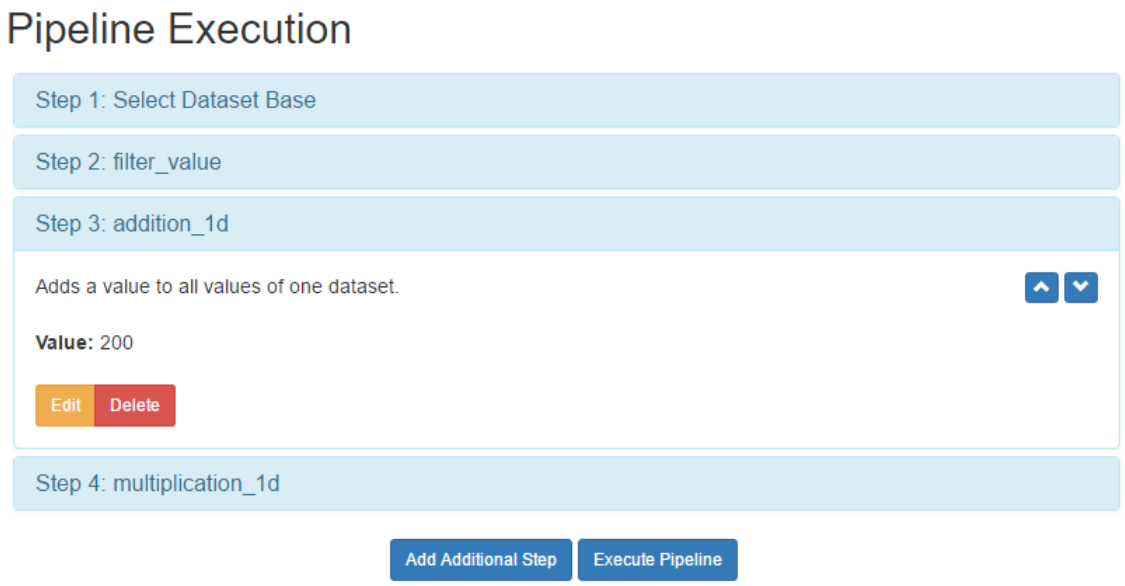

Abbildung 3.14.: Beispiel für den Pipelineaufbau

Webbrowser-Version JSON-Strings nicht mehr parsen und die Antwort des Backends nicht mehr nutzbar ist. Details zu der Leistungsfähigkeit dieses Ansatzes werden in Kapitel 5 aufgeführt.

Zusammenfassung

In diesem Kapitel wurden die verwendeten Komponenten und Datenstrukturen der Webanwendung sowie deren Zusammenspiel bei der Verwendung genauer vorgestellt. Es gibt drei Hauptkategorien an Komponenten: das Frontend, das Backend und die Datenquelle. Datenanfragen für Nutz- und Metadaten betreffen dabei hauptsächlich die Datenquelle, die Datenverarbeitung erfolgt größtenteils über das Backend. Es wurde mithilfe von unterschiedlichen Konstellationen vorgestellt, inwiefern die Komponenten angeordnet werden können.

4. Implementation

Dieses Kapitel beschreibt wichtige Stellen und Konzepte der Implementation und stellt außerdem die in Kapitel 3 kurz angeführten Komponenten des Frontends im Detail vor.

Die Anwendung wurde unter verschiedenen Systemen entwickelt. Die Entwicklung des Backends erfolgte in `R` mit der Entwicklungsumgebung *RStudio* unter Ubuntu 14.10. Das Frontend besteht aus `HTML`, `CSS` und `JavaScript` (*AngularJS* und *jQuery*). Entwickelt wurde es unter Windows 10 mithilfe von *JetBrains PhpStorm 2016.1.1(64)*.

4.1. Asynchronität von HTTP-Anfragen

Ein wichtiges Konzept für die Verwendung von *AngularJS* im Zusammenhang mit anderen Backend-Komponenten ist die Verwendung von sogenannten *Promises*, die eine bequeme Abarbeitung von asynchronen Anfragen ermöglichen. JavaScript-Applikationen laufen in einem einzigen Thread, daher sollte der Ablauf einer Anwendung nach Möglichkeit nicht für eine längere Zeit blockiert werden. Allerdings gibt es auch Funktionsaufrufe, deren Ausführungsdauer nicht abschätzbar ist. Diese Anfragen würden damit die Anwendung für unbestimmte Zeit blockieren. Zu diesen Anfragen zählen unter anderem auch die in der Web-Anwendung oft genutzten HTTP-Requests.

Diese Problematik der blockierenden Aufrufe lässt sich durch asynchrone Aufrufe umgehen. Es gibt zwei Möglichkeiten, Asynchronität umzusetzen.

1. Asynchronität durch Callbacks
2. Asynchronität durch Promises

Die Bedeutung und Notwendigkeit von Promises wird mithilfe von Listing 4.1, 4.2 und 4.5 anhand eines Beispiels gezeigt.

```
1    var response = AsyncService.doSomethingAsync(params);
2
3    // This doesn't work as expected, since response is null.
4    checkResponse( response );
```

Listing 4.1: Asynchronität ohne Promises

In Listing 4.1 wird eine asynchrone Funktion aufgerufen. Ein Beispiel hierfür wären HTTP-Requests, die Daten von einer Datenbank anfordern. Die im Listing gezeigte

Ausführung würde nicht zum gewünschten Ergebnis führen, wenn man versucht, die Antwort des asynchronen Aufrufs als Rückgabewert der dazugehörigen Funktion zu interpretieren. Da das Frontend nicht weiß, wie lange die Ausführung dauern wird, wird der Rückgabewert der Funktion mit null ausgewertet und an `response` gegeben. Auswertungen der vermeintlichen Antwort würden dann natürlich fehlschlagen. Zusätzlich wird die eigentliche Antwort, die zu einem späteren Zeitpunkt an das Frontend zurückgegeben wird, nicht weiter verarbeitet werden können.

```
1  var value;
2
3  AsyncService.doSomethingAsync( params, function( response ){
4      value = response.data.value;
5      doSomethingWithValue( value ); // OK, value is defined.
6  });
7
8  // Error, value is not defined outside the callback function:
9  doSomethingWithValue( value );
```

Listing 4.2: Asynchronität durch Callbacks

Eine Möglichkeit, die Asynchronität zu behandeln, ist die Verwendung von *Callbacks* und wird beispielhaft in Listing 4.2 skizziert. Dabei wird der asynchronen Funktion als Parameter eine Funktion mitgegeben, die nach Erhalt der Antwort aufgerufen werden soll. Der Callback-Funktion wird dann die Antwort der asynchronen Anfrage als Parameter übergeben (hier `response`). Die Callback-Funktion kann dabei eine bereits definierte Funktion sein oder auch wie im Listing 4.2 als *anonyme* Funktion (Funktion ohne Namen) übergeben werden.

Innerhalb der Funktion kann dann die Antwort über die Variable `response` ausgewertet und weiter verarbeitet werden. Das Problem dieser Lösung ist jedoch, dass außerhalb der Callback-Funktion Änderungen, die von ihr vorgenommen wurden, nicht sichtbar sind. In Listing 4.2 wird beispielsweise die Zuordnung der Antwort an die Variable `value` (Zeile 4) außerhalb der Funktion nicht erkannt. Der Aufruf in Zeile 9 würde daher erneut nicht zum gewünschten Ergebnis verhelfen. Es lässt sich bedingt Abhilfe schaffen, indem die Funktionalität, die für die Verarbeitung der Antwort asynchroner Aufrufe zuständig ist, in die Callback-Funktion ausgelagert wird.

Ein großes Problem von Callbacks wird für den (nicht seltenen) Fall sichtbar, wenn das Ergebnis eines asynchronen Aufrufs relevant für den darauf folgenden Aufruf ist. Es ist zwar möglich, den nächsten asynchronen Aufruf mit Callback in den Callback des ersten Aufrufs einzubauen, die daraus entstehende Verschachtelung wird jedoch bei mehreren aufeinanderfolgenden Aufrufen sehr unübersichtlich. Da bei asynchronen Aufrufen außerdem oftmals die Fehlerbehandlung eine wichtige Rolle spielt und in dieser Verschachtelung zusätzlich noch untergebracht werden muss, verschlechtert sich die Lesbarkeit des Codes um ein Vielfaches. Dieses Konstrukt wird daher auch *Pyramid of*

Doom genannt ([TB14], S. 308). Ein Beispiel einer *Pyramid of Doom* zeigt Listing 4.3.

```
1 AsyncService.doSomethingAsync( params, function( response ){
2     doSomethingElseAsync( response.data.value, function(
      ↪ anotherResp ){
3         doAThirdCall( anotherResp.data.value, function( thirdResp ){
4             finalEvaluation( thirdResp.data.value, function(
                  ↪ finalResp ){
5                 ...
6             });
7         });
8     });
9 });
```

Listing 4.3: Beispiel einer *Pyramid of Doom*

Um diese Verschachtelung zu vermeiden, werden in JavaScript die sogenannten *Promises* eingesetzt. Ihre grundlegende Verwendung wird in Listing 4.4 skizziert.

```
1 promise.then( onSuccess, onError );
```

Listing 4.4: Definition von Promises

Die Verarbeitung eines Promises erfolgt durch den Aufruf von `then()` und kann (auch mehrfach) an einen Aufruf einer asynchronen Funktion angehängt werden. `then()` nimmt dabei zwei Parameter entgegen, die beide für eine Funktion stehen. Der erste Parameter steht dabei für die Funktion, die die Antwort im Fall einer erfolgreichen Ausführung auswerten soll, der zweite Parameter steht für die Funktion, die im Fehlerfall beispielsweise den Nutzer auf die fehlgeschlagene Abfrage hinweisen soll. Auch hier können wieder sowohl bereits definierte als auch anonyme Funktionen verwendet werden. Listing 4.5 zeigt beide Möglichkeiten.

```
1  doSomethingElseAsync = function( param ){ ... };
2  doAThirdCall = function( param ){ ... };
3  finalEvaluation = function( param ) {...};
4
5  AsyncService.doSomethingAsync( params )
6  .then( function( response ){
7      return response.data.value;
8  }, function(error){
9      console.log('Error! Message: ' + error ) ;
10 })
11 .then( doSomethingElseAsync )
12 .then( doAThirdCall)
13 .then( finalEvaluation);
```

Listing 4.5: Asynchronität mit Promise

Hier ist die gleiche Abfolge an asynchronen Funktionsaufrufen wie in Listing 4.3 zu sehen. Im Gegensatz zu der Callback-Variante kann die unleserliche Verschachtelung an Funktionsaufrufen mit den Promises elegant aufgelöst werden, indem die Antworten sequentiell über die aneinander gereihten `then()`-Aufrufe abgearbeitet werden. In Zeile 6 bis 10 kommen hierbei zunächst zwei anonyme Funktion für die erfolgreiche Anfrage beziehungsweise den Fehlerfall zum Tragen, bevor in Zeile 11-13 auf bereits definierte Funktionen zurückgegriffen wird.

Die sequentielle Bearbeitung von asynchronen Antworten muss nicht sofort erfolgen. Rückgabewerte aus einer `then()`-Anweisung werden ebenfalls als Promise verstanden und können an anderer Stelle über die nächste `then()`-Anweisung aufgegriffen werden. Damit lassen sich Promises über mehrere Services hinweg weiterreichen. Dieses Konzept wird in der Anwendung bei jeder Anfrage an das Backend umgesetzt. Der jeweilige Sub-RequestService formatiert zunächst die Antwort, bevor der *RequestService* die formatierte Antwort weiterreicht. Das Ergebnis wird dann vom entsprechenden Service, beispielsweise dem *SessionDataService*, verarbeitet und dann schlussendlich an den Controller, der über die Nutzereingabe die Anfrage ursprünglich initiiert hat, zurückgegeben, damit dieser dem Nutzer eine Rückmeldung geben kann.

4.2. Implementation der Frontend-Komponenten

ExecutionService

Der **ExecutionService** verwaltet die meisten *unabhängigen*[1] Sub-Services und stellt für viele Controller eine zentrale Anlaufstelle für allgemeine Funktionalitäten zur Verfügung. Vor allem Anfragen an die Backend-Komponenten werden bis auf wenige Ausnahmen über die Schnittstellen des *ExecutionServices* initiiert. Über diesen Service können die Controller außerdem allgemeine Informationen der momentanen Sitzung anfordern. Dazu gehören unter anderem:

- die verfügbaren Sub-*RequestServices*,
- die zur Verfügung stehenden Sub-*VisualizationServices*,
- die durch das Backend anforderbaren NetCDF-Dateien,
- die Informationen der momentan ausgewählten Datei

Bei den vom *ExecutionService* verwalteten Komponenten handelt es sich um:

- den *RequestService* und seine Sub-Services,
- den *VisualizationService* und seine Sub-Services,

[1] Unabhängige Services sind keinem Controller direkt zuzuordnen, sondern stellen allgemeine Funktionalität zur Verfügung.

- den *SessionDataService*
- und den *DataFileService*.

RequestService

Der **RequestService** stellt die Schnittstelle des Frontends zur Kommunikation mit dem Backend oder wahlweise auch direkt mit der Datenquelle *h5serv* dar. Die Anfragen werden dazu intern über http-Requests formuliert. Die Antwort des Backends wird als JSON-String entgegen genommen und bei Bedarf formatiert. Der *RequestService* bietet den weiteren Komponenten des Frontends die Möglichkeit, ihre Anfragen über unterschiedliche "Sub-Services" auszuführen.

Zur Zeit kommen zwei *Sub-Services* zum Einsatz:

- **H5servRequestService**
  Mit diesem *Sub-Service* ist es möglich, `HTTP-Requests` direkt an *h5serv* und damit die Datenquelle selbst zu stellen. Diese Methode kommt vor allem dann zum Einsatz, wenn das Frontend Metadaten benötigt und eine Verarbeitung von Daten durch das *OpenCPU*-Backend nicht erforderlich ist.
- **OCPURequestService** Dieser Service ist dafür zuständig, Anfragen zu formulieren, die direkt an das *OpenCPU*-Backend gerichtet sind oder beispielsweise über *OpenCPU* an *h5serv* vermittelt werden sollen. Die Abarbeitung der Pipeline (siehe Abschnitt 3.4.4) erfolgt ebenfalls mithilfe des *OCPURequestServices*.

Über eine einfache Anbindungsmöglichkeit können später weitere *Sub-Services* hinzugefügt werden, sodass die Kommunikation mit zusätzlichen Komponenten, die als Backend in Frage kommen könnten, ermöglicht wird. Hierbei ist jedoch darauf zu achten, dass das Antwort-Format der zusätzlichen *Sub-Services* dem der bereits vorhandenen entspricht.

Der Nutzer kann im Wesentlichen die Verwendung eines bestimmten Sub-Services frei wählen. Die Anwendung versucht jedoch, Anfragen gezielt an den entsprechenden Sub-Service zu stellen, der für den Typ der Anfrage vorgesehen ist.

Unabhängig von der ausgewählten Backend-Komponente hat die Anbindung dieser Komponenten zur Folge, dass eine Verarbeitungszeit von Anfragen im Backend eingeplant werden muss. Diese Anfragen erfolgen durch `HTTP-Requests`. Für die Entwicklung des Frontends bedeutet dies konkret, dass sämtliche Anfragen dieser Art *asynchron* abgewickelt werden müssen. Eine asynchrone Beispielanfrage wird in Abschnitt 4.1 erläutert.

SessionDataService

Der **SessionDataService** ist für die Verwaltung der Informationen über die momentan genutzte NetCDF-Datei zuständig. Er verfügt neben den Metadaten über die Datei

und den Metadaten aus der Datei selbst (beispielsweise über die verfügbaren Datasets) zusätzlich über einige Daten (*Samples*).

Alle Informationen können getrennt geladen werden und werden daher nur bei Bedarf angefordert. Das verringert die Bearbeitungszeit von Anfragen und hält außerdem die benötigte Menge an Daten gering. Der *SessionDataService* verwaltet also hauptsächlich die Metadaten zu der angeforderten Datei. Die angesprochenen Beispieldatensätze müssen explizit vom Nutzer angefordert werden.

Eine der Hauptaufgaben des *SessionDataServices* liegt darin, für eine ausgewählte Datei die *Basis-Dimensionen* zu ermitteln. Dabei handelt es sich um "triviale", unabhängige Kenngrößen, die in der Datei festgehalten werden, wie beispielsweise die Zeit in Form einer Reihe von Zeitpunkten, an denen Messdaten erfasst wurden, oder geographische Koordinaten in Form von Längen- und Breitengraden.

Die eigentlichen Messwerte, beispielsweise für Klimadaten häufig vorkommende Werte wie Temperatur, Schneefall, Niederschlag oder Windgeschwindigkeiten, werden dann für die unterschiedlichen Kombinationen aus den Basis-Dimensionen ermittelt. Das bedeutet, dass Messwerte zur Temperatur für alle vorhandenen Zeitpunkte, Längen- und Breitengrade gespeichert werden. Fehlende Werte werden durch sogenannte *fill values* gekennzeichnet und sollten für spätere Auswertungen dementsprechend berücksichtigt werden.

Das Auslesen der eigentlichen Messwerte eines Datasets durch *h5serv* gibt ein (ggf. mehrdimensionales) Array zurück. Dieses Dataset wird durch ein Metadaten-Objekt beschrieben. Ein Beispiel hierfür findet sich in Abschnitt 3.3.1.

Die Informationen, die der *SessionDataService* zur Verfügung stellen kann, werden ausschließlich über den *ExecutionService* angefordert. Sie werden für die Zeit, in der mit einer Datei gearbeitet wird, gespeichert und bei einem Dateiwechsel verworfen, um nicht unnötig viele ungenutzte Daten zu sammeln. Bei einem Wechsel zu einer bereits verwendeten Datei müssen die Informationen erneut angefordert werden.

DataFileService

Die Hauptaufgabe des **DataFileService** ist das Bereitstellen einer Liste aller verfügbaren NetCDF-Dateien. Dafür wird beim Starten der Applikation diese Liste zunächst geladen. Zur Zeit wird im `/data/`-Verzeichnis von *h5serv* nur das Unterverzeichnis `public` eingelesen. Der Service lädt zu allen verfügbaren Dateien den Namen der Datei im Dateisystem selbst (ohne Extension) sowie die sogenannte `h5domain`. Dabei handelt es sich um den Pfad, über den die Datei bei Anfragen an *h5serv* gefunden werden kann.

Die Dateiliste des *DataFileServices* wird ebenfalls über den *ExecutionService* angefordert (→`ExecutionService.getFileList()`).

DatasetController

Der **DatasetController** (interne Bezeichnung: `PLFileListCtrl`) bildet zusammen mit seinem HTML-Template `/templates/modal/tpl_filelist.html` den modalen Dialog zur Auswahl einer NetCDF-Datei, eines Datasets und gegebenenfalls auch expliziter Werte unabhängiger Dimensionen wie Zeitpunkt oder Längen-/Breitengrad.

Der Nutzer kann hierbei zunächst eine Datei aus der Liste verfügbarer NetCDF-Dateien auswählen. Die weitere Auswahl umfassen neben dem Dataset auch andere Dimensionen, die bei späteren Visualisierungsvorgängen auf den entsprechenden Achsen abgebildet werden. Außerdem hat der Nutzer die Möglichkeit, die zu ladenden Werte mithilfe von Vorgaben des Datasets wie `scale_factor` und `add_offset` anzupassen.

Das Ergebnis dieses Auswahldialogs ist die Basis-Datensatz-Struktur, die in Abschnitt 3.3.2 detailliert vorgestellt wird. Diese Datenstruktur wird in fast jedem Schritt bis zur abschließenden Visualisierung benötigt.

PipelineFunctionController

Der **PipelineFunctionController** (interne Bezeichnung: `PLFunctionCtrl`) stellt mit dem HTML-Template `/templates/modal/tpl_pl_function.html` den modalen Dialog zur Auswahl einer Pipeline-Funktion sowie eventuell benötigter Parameter zur Verfügung.

Sie lädt die verfügbare Funktionsliste aus dem PipelineService und gleicht diese mit dem ausgewählten Basis-Datensatz ab, da nicht jede Funktion auf jede Art von (Sub-)Dataset anwendbar ist. Das abschließende Bestätigen des Auswahldialogs fügt die ausgewählte Funktion sowie die eingegebenen Parameter zur momentanen Pipeline hinzu.

PipelineService

Der **PipelineService** ist für die Verwaltung einer ausführbaren Pipeline zuständig. Diese Pipeline kann zahlreiche Schritte umfassen, die dann an das Backend gesendet werden, wo sie von *OpenCPU* interpretiert und ausgeführt werden.

Vor dem Absenden der Pipeline muss der Service alle Funktions- und Parametereingaben validieren. Außerdem muss die interne Datenstruktur, die die Pipeline beschreibt, in einen JSON-String umgebaut werden, der von der *OpenCPU*-Bibliothek im Backend interpretiert werden kann.

Neben dem Ausführen der Pipeline, also dem Absenden der Pipeline und dem Erhalt der Antwort, ist diese Komponente auch dafür zuständig, den eigentlichen Aufbau der

Pipeline zu verwalten. So können einzelne Funktionen in der Pipeline nach vorne oder hinten verschoben werden, sie können angepasst oder gelöscht werden und weitere Schritte der Pipeline hinzugefügt werden. Der Ablauf einer Pipelineausführung wird in Abschnitt 3.4.4 detailliert dargestellt.

PreviewController

Die Hauptaufgabe des **PreviewController** liegt darin, die unterschiedlichen Benutzereingaben zu vereinen, um als Ergebnis eine erste Vorschau der aus der NetCDF-Datei gelesenen Daten zu erhalten. Für den Controller ist daher vor allem die Auswahl des Basis-Datensatzes und der Visualisierungsart wichtig.

Diese Auswahl erfolgt stufenweise über das dazugehörige HTML-Template `/templates/preview.html`, da Abhängigkeiten zwischen den Benutzereingaben existieren. Die stufenweise Auswahl umfasst:

1. Auswahl eines Basis-Datensatzes (sowie seines Typs)
2. Auswahl einer Visualisierungsbibliothek
3. Auswahl einer Visualisierungsmethode

Neben der Visualisierung ist der *PreviewController* auch für die Abbildung der in Abschnitt 4.2 vorgestellten Snapshots verantwortlich.

VisualizationService

Der **VisualizationService** ist dafür zuständig, die "rohen" Antwort-Daten des Backends in eine grafische Repräsentation umzuwandeln. Dafür erhält der Nutzer je nach ausgewähltem Dataset sowie den ausgewählten Dimensionen eine Liste an verfügbaren Visualisierungsmöglichkeiten (Liniendiagramm, Balkendiagramm, Heatmap...).

Genau wie der *RequestService* stellt auch der *VisualizationService* verschiedene Sub-Services zur Verfügung, die ihrerseits die unterschiedlichen grafischen Repräsentationen zur Verfügung stellen. Zur Zeit sind zwei Sub-Services implementiert:

- **FlotchartVisualizationService**
  Dieser Sub-Service basiert auf der Bibliothek *flotchart*[2]. Diese Bibliothek bietet eine Vielzahl von interaktiven 2D-Visualisierungsmöglichkeiten an. So können per `MouseOver` einzelne Datenpunkte im Diagramm ausgelesen werden. Außerdem können Bereiche im Diagramm markiert werden, an die herangezoomt werden soll. Per Doppelklick auf das Diagramm wird es wieder auf die ursprüngliche Ansicht zurückgesetzt.
  Der Sub-Service stellt zur Zeit die beiden Visualisierungsmöglichkeiten "Linien-" und "Balkendiagramm" zur Verfügung.

[2] `http://www.flotcharts.org/`

- **HeatmapVisualizationService**
  Dieser Sub-Service dient dazu, 3D-Visualisierungen darzustellen (X-Achse, Y-Achse, Farbe der Punkte). Er basiert auf der Bibliothek *heatmap.js*[3]. Der Service wurde implementiert, um die grafische Darstellung größerer Datensätze zu testen. Die Leistungsfähigkeit der zugrunde liegenden Bibliothek wurde vom Entwickler mit 40.000-60.000 Datenpunkten angegeben, in Tests funktionierte die Visualisierung auch bei mehr als 115.000 Datenpunkten ohne Probleme. Die Bibliothek selbst bietet u.a. die Möglichkeit, die Farbwerte frei konfigurieren zu können. Der Sub-Service verwendet zur Zeit allerdings eine festgelegte farbliche Repräsentation der Daten. Mit Auswahl der richtigen Dimensionen als Achsen können sich trotzdem aussagekräftige Darstellungen ergeben, ein Beispiel zeigt Abbildung 4.1. Features wie etwa eine Farblegende oder die Beschriftung der Achsen sind zur Zeit noch nicht verbaut.
  Der Sub-Service stellt zur Zeit die Möglichkeit "Heatmap" für die Visualisierung bereit.

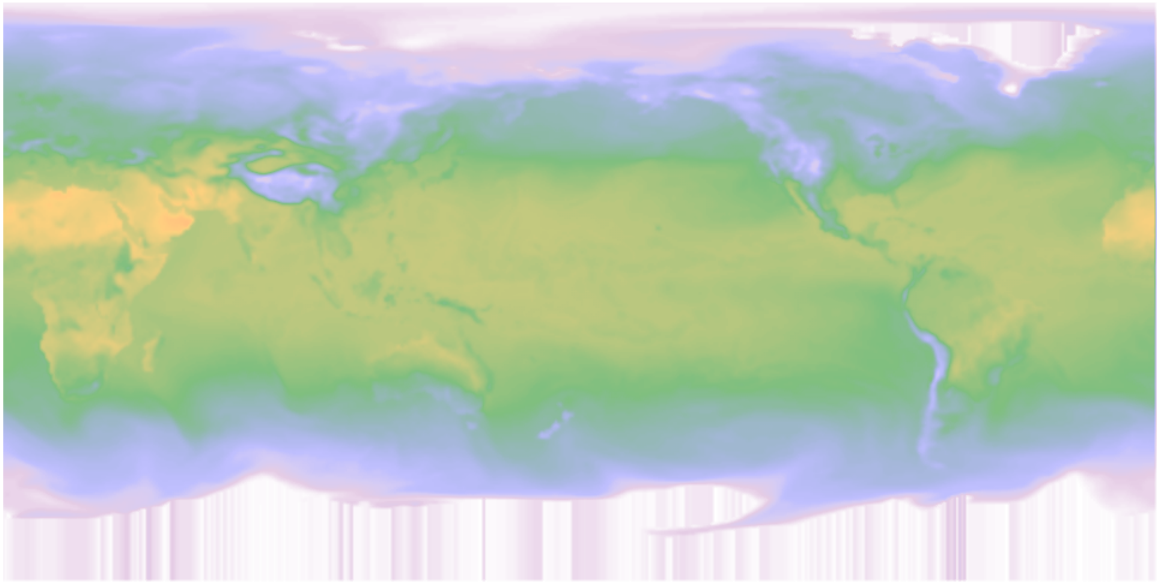

Abbildung 4.1.: Beispiel einer Heatmap-Visualisierung (Temperatur weltweit, 19.09.1996). Achsenbeschriftung und Legende sind in der Testversion noch nicht implementiert.

Der *VisualizationService* kann ebenfalls um weitere Sub-Services ergänzt werden. Dazu müssen die neuen Services:

- Im *VisualizationService* registriert sein,
- Eine Methode zur Verfügung stellen, die die übergebenen Rohdaten in eine für die verwendete Bibliothek gültige Struktur formatiert (`formatRawData()`),

[3] `https://www.patrick-wied.at/static/heatmapjs/`

- Eine Methode zur Verfügung stellen, die die Visualisierung durchführt (`visualize()`) und
- Eine Liste an Visualisierungstypen angeben, die mit ihnen dargestellt werden können.

PerformanceService

Der **PerformanceService** protokolliert unterschiedliche Kriterien zu allen HTTP-Requests, die vom Frontend ausgeführt werden. Zu den Kriterien zählen:

- Verwendeter Sub-RequestService
- Die an das Backend übermittelte URL (der sogenannte `request_path`)
- Start- (Absenden der Anfrage) und Endzeitpunkt (Erhalt der Antwort) der Anfrage
- Dauer der Anfrage
- Größe der Antwort (in kB)
- Status des HTTP-Requests (`COMPLETE`, `CANCELED` und `ONGOING`)

Diese Komponente arbeitet eng mit dem *AlertService* zusammen, damit der Status einer Anfrage direkt nach dem Abschluss dem Nutzer sichtbar gemacht werden kann.

Außerdem speichert der Service alle durchgeführten Anfragen, sodass diese über eine Tabelle, die über die Hauptseite der Anwendung erreichbar ist, eingesehen werden können. Hier werden außerdem auch die aufsummierte Zeit der Anfragen sowie die Gesamtgröße aller erhaltenen Antworten angezeigt. Es ist außerdem möglich, sich die Messergebnisse von einem auswählbaren Teil der Anfragen berechnen zu lassen.

LoggingService und AlertService

Diese beiden Services sind bei gestellten HTTP-Anfragen für das Feedback an den Nutzer sowie für etwaige Fehler bei der Ausführung der Anwendung zuständig. Während der **LoggingService** unterstützende Debug-Ausgaben für die JavaScript-Konsole bereitstellt, zeigt der **AlertService** wichtige Informationen, beispielsweise über erfolgreiche oder fehlgeschlagene Anfragen, direkt in der Benutzeroberfläche an.

Die Ausgaben des *LoggingServices* können durch die unterschiedlichen `OUTPUT_LEVEL` des *ConfigServices* gesteuert werden:

- `OUTPUT_LEVEL_NONE`
  Es werden keine Konsolenausgaben getätigt.
- `OUTPUT_LEVEL_ERROR`
  Die Anwendung informiert über die Konsole nur über Fehler im Ablauf.

- `OUTPUT_LEVEL_NOTICE`
  Es werden zusätzlich zu `OUTPUT_LEVEL_ERROR` Meldungen ausgegeben, die den Benutzer über den Status der Anwendung informieren (Beispiel: Initiieren des Ladens der Datei-Liste sowie den erfolgreichen Abschluss dieses Vorgangs).

- `OUTPUT_LEVEL_DEBUG`
  Die Anwendung protokolliert neben den Ausgaben von `OUTPUT_LEVEL_NOTICE` detailliertere Informationen zu Backend-Anfragen. Zusätzlich wird der Befehl `LoggingService.JSONOutput()` aktiv geschaltet, der übergebene Objekte lesbar als `JSON`-String in der Konsole ausgibt.

- `OUTPUT_LEVEL_DEBUG_EXTRA`
  Neben den bereits aufgeführten Protokollausgaben werden mit diesem Level `HTML`-Elemente in der Anwendung selber aktiviert, die per *Two-way-data-binding* (siehe Abschnitt 2.1.2) Objekte formatiert als `JSON`-String ausgeben.

Die beiden Debug-Level werden normalerweise nur für die Entwicklung beziehungsweise Erweiterung der Webanwendung benötigt. Bei der normalen Verwendung der Anwendung sind die Informationen des `OUTPUT_LEVEL_NOTICE` ausreichend.

Zusätzlich wird der Nutzer durch den zweiten Service, den *AlertService*, über verschiedene Abläufe direkt in der Anwendung durch kleine Dialoge am unteren Bildschirmrand informiert. Sie können manuell ausgeblendet werden, verfügen aber größtenteils auch über einen `timeout`-Zähler (in ms), der die Meldungen automatisch nach Ablauf dieser Zeit wieder ausblendet. Diese Dialoge sind nicht mit dem Output-Level gekoppelt.

Daneben hat der *AlertService* noch die Aufgabe, unterschiedliche, einfache Dialoge bereitzustellen,die die Eingaben des Nutzers abfragen. Dabei wird unter anderem die Bestätigung eines Arbeitsschrittes oder die Eingabe eines Zeichenkette (beispielsweise für die Benennung eines *Snapshots*, siehe Abschnitt 4.2) erwartet.

ConfigService

Der **ConfigService** verwaltet wesentliche Konfigurationsoptionen der Anwendung. Neben den eben beschriebenen Ausgabe-Optionen des *LoggingServices* verwaltet dieser Service auch die Anbindung per `IPv4`-Adresse, über die die Backend-Komponenten *h5serv* und *OpenCPU* erreichbar sind.

SnapshotService

Der **SnapshotService** stellt eine Erweiterung des *VisualizationServices* dar. Er bietet dem Nutzer die Möglichkeit, eine gerade ausgegebene Visualisierung zwischenzuspeichern. Diese gespeicherte Version der Visualisierung kann dann beispielsweise zu einem späteren Zeitpunkt und zu einer anderen Visualisierung wieder geladen werden, um einen direkten

grafischen Vergleich zu erhalten.

Der *SnapshotService* speichert ein exaktes Abbild der Vorlage. Das bedeutet, dass nicht etwa ein temporärer Screenshot der Vorlage gespeichert wird, sondern auch zusätzliche, interaktive Funktionen beim Laden wieder zugänglich gemacht werden. Dazu zählen unter anderem eine Zoom-Möglichkeit und die dynamische Anzeige von Datenpunkten.

HelperService

Der **HelperService** umfasst eine Sammlung von wichtigen Funktionen, die häufig an verschiedenen Stellen innerhalb des Frontends benötigt werden. Hierzu zählt neben diversen Formatierungsfunktionen einer URL vor allem die Möglichkeit, in einem Array von Objekten nach dem Objekt zu suchen, das für ein vorgegebenes Attribut einen bestimmten Wert hält.

4.3. H5servBackend

H5servBackend ist eine Bibliothek, die auf `R` basiert und verschiedene Funktionalität für die Backendkomponente *OpenCPU* anbietet. Dazu gehört die Verarbeitung von Anfragen, die vom Frontend an das Backend gestellt werden, aber auch die Möglichkeit, selbst HTTP-Requests an die Datenquelle zu stellen.

Der Funktionsumfang lässt sich in drei Bereiche unterteilen:

- Weiterleitung von HTTP-Requests und -Antworten,
- Ausführung von CDO-Anfragen und
- Abarbeitung einer Pipeline

Die Weiterleitung der HTTP-Requests nimmt die von *OpenCPU* übergebenen Parameter entgegen und baut mit Hilfe eigener Konfigurationswerte einen HTTP-Requests für *h5serv* zusammen. Diese Anfrage wird dann an *h5serv* weitergereicht. Die Antwort der Datenquelle wird abschließend zurückgegeben[4].

Bei der Ausführung von CDO-Anfragen nimmt die Bibliothek einige Formatierungen am übergebenen Parameter, der den CDO-befehl enthält, vor, bevor dieser Befehl per *System Call* ausgeführt wird. Die Antwort besteht aus den Ausgaben der CDO-Ausführung und wird abschließend ähnlich wie bei der Weiterleitung der HTTP-Requests zurückgegeben. Das eigentliche Ergebnis der CDO-Ausführung besteht aus einer neuen NetCDF-Datei. In der aktuellen Version wird das Ergebnis verworfen, kann jedoch später direkt in das entsprechende `/data/`-Verzeichnis von *h5serv* geschrieben werden, wodurch es für

[4] *OpenCPU* speichert den Rückgabewert der Funktion als Session ab und beantwortet die ursprüngliche Anfrage des Frontends mit dem dazugehörigen *session key*.

darauffolgende Analysen sofort verfügbar ist.

Bei der Abarbeitung einer Pipeline wird der entsprechenden Funktion die Pipeline als JSON-String übergeben. Die Bibliothek parst den String und erhält dadurch zwei Datenstrukturen für die auszuführende Funktionsliste sowie die dazugehörigen Parameter. Die Funktionsliste wird dann mit den entsprechenden Parametern der Reihe nach abgearbeitet, bevor schlussendlich das Ergebnis der letzten Funktion zurückgegeben wird.

Zusammenfassung

In diesem Kapitel wurde mit den asynchronen HTTP-Requests eine der wichtigsten Kernfunktionalitäten der Webanwendung erläutert. Dazu wurden Implementationsdetails zu den Frontend-Komponenten ausführlich erklärt. Die Details des Anwendungsaufbaus sind nun insofern bekannt, dass im Folgenden Beispielmessungen zur Ermittlung der Leistungscharakteristika zu den einzelnen Komponenten analysiert werden können.

5. Evaluation

Dieses Kapitel wertet die Messergebnisse zur Performanz der Webanwendung aus. Dazu wird zunächst die Testumgebung in Abschnitt 5.1 beschrieben und anschließend in Abschnitt 5.2 die Vorgehensweise bei den Messungen vorgestellt. Danach werden die Messergebnisse der einzelnen Test (siehe Abschnitt 5.3) präsentiert, bevor abschließend in Abschnitt 5.4 die Ergebnisse ausgewertet werden.

5.1. Testumgebung

Backend und Datenquelle sind auf einem Forschungscluster der Arbeitsgruppe *Wissenschaftliches Rechnen* der Universität Hamburg installiert, welcher im DKRZ in Hamburg, Deutschland steht:

- 4 Sockel-System, AMD Opteron (™) Processor 6344
- 256 GB RAM
- Ubuntu 14.04 LTS
- Daten werden per NFS gespeichert
- Anbindung über GB-Ethernet

Das Frontend wurde lokal aufgesetzt, kann jedoch unter jedem beliebigen Webserver bereitgestellt werden, da JavaScript hier ausschließlich clientseitig und damit auf dem Rechner des Nutzers ausgeführt wird. Die Verbindung zwischen lokalem Frontend und serverseitigen Komponenten erfolgte per SSH-Tunnel über eine 50-Mbit-Anbindung. Dabei sollte beachtet werden, dass die Nutzung eines SSH-Tunnels zusätzliche CPU-Zeit beansprucht.

5.2. Vorgehensweise

Ziel der Evaluation ist es, neben wichtigen Parametern in Bezug auf die client- und serverseitige Auslastung vor allem die Ausführungszeit zu untersuchen, die ein Benutzer abwarten muss, nachdem er eine Anfrage über die Benutzeroberfläche im Fronend gestellt hat. Abbildung 5.1 skizziert die verschiedenen Komponenten, die bei der Abarbeitung der Anfragen beteiligt sind. Der Fokus der Tests liegt dabei auf dem Frontend und der Datenquelle *h5serv*. Anschließend werden weitere Tests für Backend-Komponenten

durchgeführt. Man erkennt für den skizzierten Aufbau, dass die Ausführungszeit in mehrere Abschnitte unterteilt werden kann.

Abbildung 5.1.: Skizzierter Versuchsaufbau mit relevanten Komponenten

Es ergibt sich daher für die Gesamtzeit t einer Verarbeitung einer Anfrage:

$$t = t_{client} + t_{net} + t_{serv} + t_{InOut},$$

Dabei stehen t_{client} für die Ausführungszeit auf der Clientseite und damit im Frontend, t_{net} für die benötigte Zeit für die Netzwerkkommunikation, t_{serv} für die Verarbeitungszeit des Servers sowie t_{InOut} für die benötigte Zeit bei Festplattenzugriffen (I/O). Um die einzelnen Größen ermitteln zu können, werden unterschiedliche Tests ausgeführt. Diese werden im Folgenden vorgestellt.

Die Tests werden dabei auf unterschiedlichen Testdateien ausgeführt. Neben der zeitlichen Komponente soll mit Hilfe des Tools `Performance Co-Pilot` (kurz `pcp`[1] genannt) die serverseitige Auslastung untersucht werden. Die Untersuchungen sollen Aufschluss darüber geben, inwiefern die einzelnen Komponenten für den vorgestellten Ansatz geeignet sind. Dafür werden zwei unterschiedliche Anfragen ausgeführt:

- Abfrage der Daten zu einer 2D-Visualisierung
- Abfrage der Daten zu einer Heatmap-Visualisierung

Für die Messungen werden drei unterschiedliche Testdateien genutzt:

- `atls-1Var` (257 MB, ein Dataset, Dimension: $1096 \cdot 241 \cdot 480$)
- `atls-Full` (1,8 GB, sieben Datasets, Dimension: $1096 \cdot 241 \cdot 480$)
- `ICON`[2] (57,7 GB, 37 Datasets, Dimension: $123 \cdot 384 \cdot 768$ ($\cdot 11$ ggf., zusätzliche Dimension)

`atls-1Var` umfasst nur eines der Datasets aus `atls-Full` und beschreibt daher nur eine Teilmenge der zweiten Testdatei. Die dritte Testdatei `ICON` enthält andere Daten als die beiden anderen Dateien und ist daher von der Nutzdatenstruktur her abweichend aufgebaut. So haben einige Datasets aus der `ICON`-Datei beispielsweise vier Dimensionen.

[1] `http://pcp.io/index.html`

[2] `ICON` steht für *Icosahedral non-hydrostatic* und beschreibt das *Icosahedral non-hydrostatic general circulation model*, siehe auch: `http://www.mpimet.mpg.de/en/wissenschaft/modelle/icon.html`

Neben dem Faktor Zeit sind weitere verschiedene Faktoren interessant. Für die Backend-Komponenten lassen sich CPU-Auslastung, Arbeitsspeicherverbrauch, Netzwerkauslastung und I/O-Zugriffe untersuchen, weitere wichtige Informationen kann die Web-Oberfläche über den *PerformanceService* aus Abschnitt 4.2 ermitteln. Dazu zählen Dauer einer Anfrage (Absenden einer Anfrage bis Erhalt einer Antwort) sowie Größe der Antwort.

Es sollte außerdem beachtet werden, inwiefern vor allem die Ansprüche eines Benutzers an die Verarbeitungszeit erfüllt werden können. Dabei muss zwischen der Reaktionsgeschwindigkeit der Benutzeroberfläche und Verarbeitung von Daten unterschieden werden. Während die Reaktionsgeschwindigkeit der Oberfläche, beispielsweise beim Bereitstellen der Daten für kommende Auswahlfelder nach einer Benutzereingabe, nicht länger als zwei Sekunden dauern sollte, kann die Verarbeitung von Daten bei entsprechendem Hinweis an den Benutzer auch 20 bis 30 Sekunden in Anspruch nehmen.

Experiment 1: Verarbeitungszeit des Servers

Im ersten Test wird mit t_{serv} die benötigte **Bearbeitungzeit des Servers** ermittelt. Dazu werden Anfragen wie in Abbildung 5.2 gezeigt per `cURL` direkt auf dem Server gestellt, um die Frontend-Komponente und die Netzwerk-Latenz ignorieren zu können. Zusätzlich wird die benötigte Datei auf dem Server in `/dev/shm` und damit im Arbeitsspeicher abgelegt. `shm` steht für *shared memory* und wird unter Linux mit dem Dateisystem `tmpfs` (*temporary file system*) verwendet. Es verhält sich wie ein Dateisystem, verwendet jedoch keinen persistenten Speicher wie beispielsweise eine Festplatte, sondern nutzt den Arbeitsspeicher, um temporär Daten speichern zu können. Dadurch kommen keine Festplattenzugriffe zustande. Die Ausgabe von `cURL` erfolgt nach `/dev/null`, wird also direkt verworfen. Es ergibt sich bei dieser Backend-Messung für die serverseitige Bearbeitungszeit:

$$
\begin{aligned}
t_1 &= t_{serv} + t_{InOut} \\
\Rightarrow t_1 &= t_{serv} + 0 \\
\Rightarrow t_1 &= t_{serv}
\end{aligned}
$$

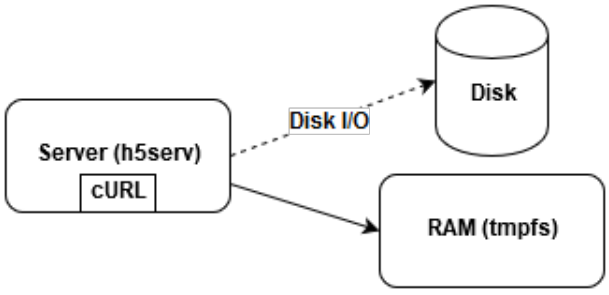

Abbildung 5.2.: Aufbau zur Ermittlung der Bearbeitungszeit des Servers ohne I/O-Zugriffe durch Verwendung von `tmpfs`

Experiment 2: I/O-Zugriffe

Auf den Ergebnissen des ersten Tests aufbauend kann im zweiten Test die benötigte **Zeit für I/O-Zugriffe** ermittelt werden. Der Testaufbau ist in Abbildung 5.3 dargestellt und entspricht dem Ablauf vom vorherigen Test. Der Unterschied ist jedoch, dass die Datei bei diesem Test auf der Festplatte gespeichert ist und somit I/O-Zugriffe entstehen. Da *h5serv* selber einen Cache für Anfragen aufbaut, ist es wichtig, zwischen den Tests sowohl den *Page Cache* von Linux zu leeren als auch *h5serv* neu zu starten. Für die I/O-Zugriffszeit ergibt sich dann:

$$
\begin{aligned}
& t_2 = t_{serv} + t_{InOut} \\
\Rightarrow\ & t_2 = t_1 + t_{InOut} \\
\Rightarrow\ & t_{InOut} = t_2 - t_1
\end{aligned}
$$

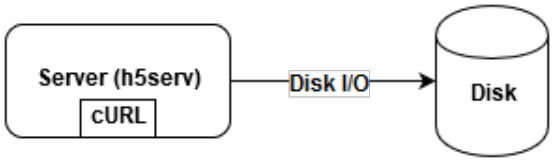

Abbildung 5.3.: Aufbau zur Ermittlung der benötigten I/O-Zugriffszeiten

Experiment 3: Netzwerkkommunikation

Im dritten Test wird auf Basis der vorherigen Ergebnisse die benötigte **Zeit für die Netzwerkkommunikation** errechnet. Dadurch lässt sich beispielsweise abschätzen, ob das Netzwerk ausgelastet werden kann. Abbildung 5.4 zeigt den Aufbau, bei dem die Anfrage wie zuvor über `cURL` gestellt wird. Bei diesem Test wird `cURL` jedoch vom Client aufgerufen, wodurch die Anfrage über das Netzwerk an den Server weitergeleitet wird. Für diesen Aufbau ergibt sich die folgende Zeit t_3 und daraus resultierend die benötigte Zeit für die Netzwerkkommunikation:

$$
\begin{aligned}
t_3 &= t_{net} + t_{serv} + t_{InOut} \\
&\Rightarrow t_3 = t_{net} + t_2 \\
&\Rightarrow t_{net} = t_3 - t_2
\end{aligned}
$$

Abbildung 5.4.: Aufbau zur Ermittlung der benötigten Zeit für Netzwerkkommunikation. `cURL` wird clientseitig ausgeführt.

Experiment 4: Verarbeitungszeit der Weboberfläche

Der vierte Test untersucht aufgrund der vorhergegangenen Ergebnisse einerseits die **Verarbeitungszeit des Frontends** sowie die **insgesamt benötigte Zeit einer Anfrage**. Dadurch lässt sich der Overhead von JavaScript beziehungsweise *AngularJS* untersuchen. Hierfür wird der in Abbildung 5.1 bereits gezeigte Aufbau verwendet, bei der die Abfragen direkt über die Benutzeroberfläche und nicht per `cURL` abgeschickt werden. Die Abläufe in diesem Test stellen typische Anwendungsfälle für den Benutzer dar. Für die Verarbeitungszeit des Frontends ergibt sich dann im Idealfall:

$$\begin{aligned} t &= t_{client} + t_{net} + t_{serv} + t_{InOut} \\ &\Rightarrow t = t_{client} + t_3 \\ &\Rightarrow t_{client} = t - t_3 \end{aligned}$$

Experiment 5: OpenCPU

Während die vorherigen vier Tests sich auf den Aufbau aus Abbildung 5.1 und damit vor allem auf die beiden Komponenten Frontend und *h5serv* fokussierten, untersucht dieser Test, wie leistungsfähig eigene `R`-Bibliotheken von *OpenCPU* eingesetzt werden können. Dieser Test betrifft daher vor allem die zuvor noch nicht berücksichtigte Backend-Komponente. Dabei wird der Overhead von OpenCPU im Vergleich zur direkten Ausführung über die `R`-Konsole untersucht. Die benötigte Datei wird auch bei diesem Test in `/dev/shm/` abgelegt.

5.3. Messergebnisse

Experiment 1: Verarbeitungszeit des Servers

Für die Verarbeitungszeit des Servers ergeben sich die Messergebnisse aus Tabelle 5.1 für die Datenanfrage der 2D-Visualisierung sowie aus Tabelle 5.2 für die Datenanfrage der Heatmap-Visualisierung.

Man erkennt, dass die Zugriffszeiten bei beiden `atls`-Dateien sehr ähnlich sind. Die jeweiligen Anfragen betreffen unterschiedliche Datasets, daher kommt beispielsweise aufgrund der unterschiedlichen Präzision der Daten eine unterschiedliche Gesamtgröße zustande. Im direkten Vergleich zur `ICON`-Datei, deren Abfrage eine größere Antwort erzeugt, gibt es noch den gravierenden Unterschied, dass in der `ICON`-Datei die Präzision über Nachkommastellen über einen doppelt so großen Datentyp erzielt wird: In den `atls`-Dateien werden die Daten als `H5T_STD_I16LE` (16-bit), in der `ICON`-Datei als `H5T_IEEE_F32LE` (32-bit) gespeichert.

Auch die wesentlich größere Antwort bei der Anfrage zur Heatmap-Visualisierung lässt sich auf die unterschiedlichen Datentypen zurückführen. Die um Faktor 6 größere Antwort der `ICON`-Datei ergibt sich ungefähr durch die um Faktor 2,5 größere Anzahl an Datenpunkten, die zusätzlich doppelt so viel Speicher benötigen.

	atls-1Var	**atls-Full**	**ICON**
Verarbeitungszeit	0,387 s	0,396 s	0,235 s
Gesamtgröße der Antwort (JSON)	11,6 kB	13 kB	18,8 kB
Anzahl Datenpunkte	1096	1096	768

Tabelle 5.1.: Ergebnisse zu Experiment 1, Datenabfrage für eine 2D-Visualisierung

	atls-1Var	**atls-Full**	**ICON**
Verarbeitungszeit	0,297 s	0,299 s	0,707 s
Gesamtgröße der Antwort (JSON)	760 kB	833 kB	5,1 MB
Anzahl Datenpunkte	115680	115680	294912

Tabelle 5.2.: Ergebnisse zu Experiment 1, Datenabfrage für eine Heatmap-Visualisierung

Experiment 2: I/O-Zugriffe

Die Messergebnisse zum zweiten Experiment sind in Abbildung 5.5 für die 2D-Visualisierung und in Abbildung 5.6 für die Heatmap-Visualisierung im direkten Vergleich zu den Ergebnissen aus dem ersten Experiment gezeigt. Im Gegensatz zum vorherigen Experiment werden die NetCDF-Dateien von der Festplatte gelesen, es fällt also eine zusätzliche Bearbeitungszeit durch I/O-Zugriffe an. Es werden die gleichen Abfragen wie in Experiment 1 verwendet, dementsprechend ändern sich weder die Gesamtgröße der Antwort noch die Anzahl der Datenpunkte.

Durch die I/O-Zugriffe gibt es bei der 2D-Visualisierung (Abbildung 5.5) vor allem bei den beiden `atls`-Dateien einen stark gestiegenen Zeitbedarf. Das lässt sich vor allem auf die Struktur der Daten in der Datei sowie die gewählte Anfrage zurückführen. Die Daten sind für alle Testdateien in multidimensionalen Arrays mit der folgenden Reihenfolge an Dimensionen gespeichert: Zeit, (Level,) Breitengrad, Längengrad. Dabei wurden für die `atls`-Dateien im multidimensionalen Datenarray explizit einzelne Werte für Breiten- und Längengrad für alle verfügbaren Zeitpunkte gewählt. Die Abfrage der `ICON`-Datei wurde für alle verfügbaren Längengrad-Werte mit festen Werten für die anderen Dimensionen aufgebaut, damit die Menge der ausgewählten Datenpunkte in etwa mit der der `atls`-Dateien vergleichbar ist. Durch diese unterschiedlichen Zugriffsmuster steigt der Zeitbedarf durch I/O-Zugriffe bei der `ICON`-Datei nur unwesentlich.

Bei der Heatmap-Visualisierung (Abbildung 5.6) sind die Anfragen trotz einer größeren angeforderten Datenmenge einfacher aufgebaut. Für alle Testdateien werden alle Werte der

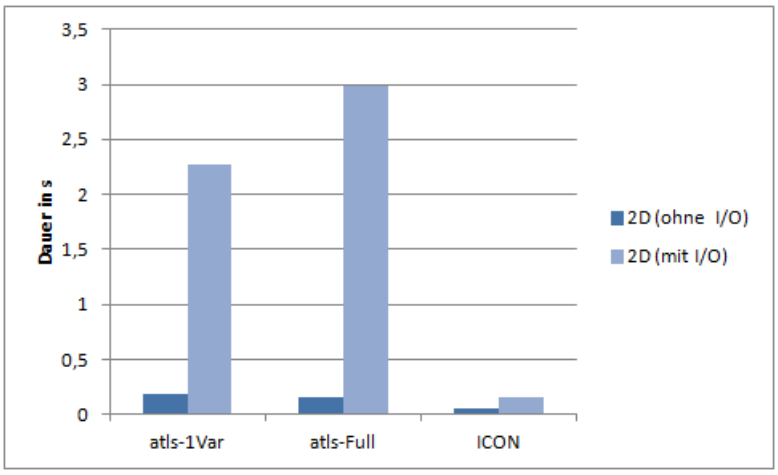

Abbildung 5.5.: Vergleich von Experiment 1 und 2 (2D-Visualisierung)

Breitengrad- und Längengrad-Dimension für einen festen Zeitpunkt (und gegebenenfalls Level-Wert) angefordert. Daher ergibt sich durch die I/O-Zugriffe nur ein geringer Mehrwert für den Zeitbedarf der ausgeführten Anfragen.

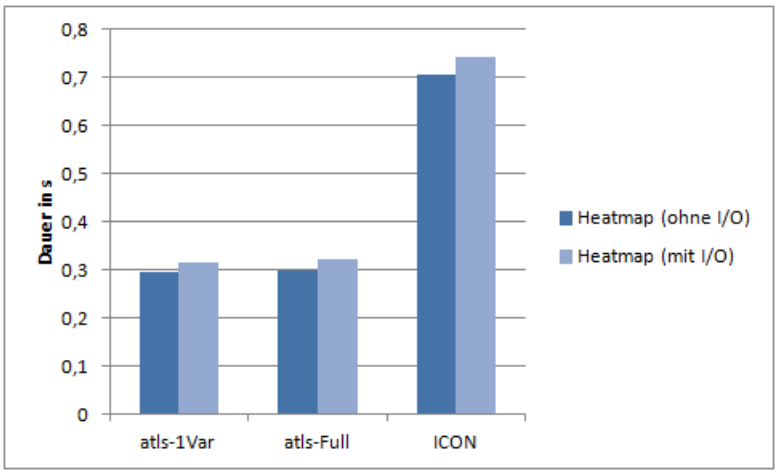

Abbildung 5.6.: Vergleich von Experiment 1 und 2 (Heatmap)

Experiment 3: Netzwerkkommunikation

Im dritten Experiment wurde die Auswirkung der Netzwerkkommunikation untersucht, wenn die vorherigen Anfragen von einem Client an den Server gestellt werden. Abbildung 5.7 und 5.8 zeigen den zusätzlichen Zeitbedarf durch die Netzwerkkommunikation, der jedoch je nach Testdatei stark variieren kann. Es zeigen sich vor allem für die 2D-Visualisierung (Abbildung 5.7) große Unterschiede. Während bei den `atls`-Dateien der Großteil des Mehraufwands im Vergleich zum Ergebnis des ersten Experiments auf die

I/O-Zugriffe aus Experiment 2 zurückzuführen sind, beansprucht die Netzwerkkommunikation bei der Anfrage für die `ICON`-Datei den Großteil der Ausführungszeit.

Die hier gezeigten Messungen für die Netzwerkkommunikation wurden nicht mehr mit `cURL` durchgeführt. Stattdessen wurde `wget`, welches in der Ubuntu-Standardinstallation bereits enthalten ist, verwendet. Zuvor durchgeführte Messungen mit `cURL` offenbarten große Inkonsistenzen der Messergebnisse, bei denen beispielsweise die eigentlich zeitaufwendigere Bearbeitung der Anfrage durch ein JavaScript-Frontend wesentlich schneller durchgeführt wurde als die Bearbeitung durch `cURL`. Zusätzlich ergab eine `cURL`-Messung für die 2D-Visualisierung der `atls-Full`-Datei unter anderem eine Bearbeitungszeit von knapp 15 Sekunden (ohne Netzwerkkommunikation), während die darauffolgende Abfrage mit Netzwerkkommunikation nur knapp 3 Sekunden dauerte. Für die vorherigen Experimente ergaben sich für die Nutzung von `wget` ähnliche Ergebnisse wie bei der Verwendung von `cURL`.

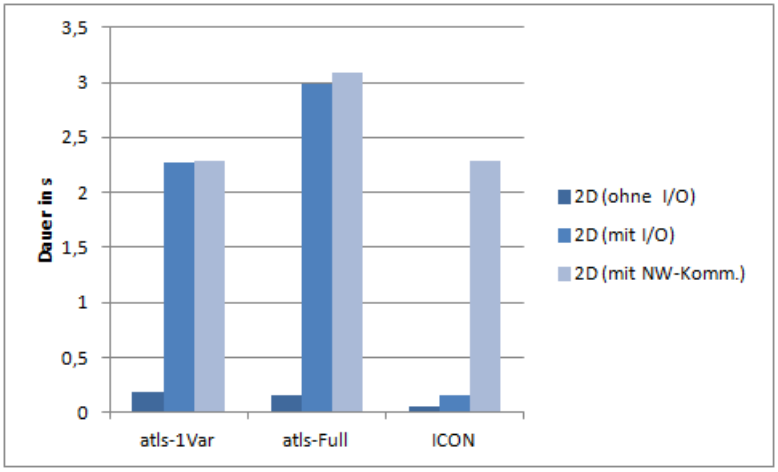

Abbildung 5.7.: Vergleich von Experiment 1, 2 und 3 (2D-Visualisierung)

Experiment 4: Verarbeitungszeit der Weboberfläche

Für die Verarbeitungszeit der Weboberfläche wird neben den beiden bekannten Anfragen zur 2D- beziehungsweise Heatmap-Visualisierung auch die Auswahl eines Basis-Datasets untersucht. Dieser Arbeitsschritt besteht aus mehreren Einzelabfragen und kann daher ohne großen Aufwand nicht über `curl` ausgeführt werden.

Die Auswahl eines Basis-Datasets stellt einen häufig vorkommenden Arbeitsschritt dar und beansprucht vor allem die Datenquelle, in diesem Fall also *h5serv*. Dabei werden mehrere Anfragen im Wechsel mit Nutzereingaben ausgeführt. Der Ablauf entspricht der Abarbeitung des in Abbildung 3.12 gezeigten Dialogs. Die abzufragenden Datenmengen sind mit meist wenigen Kilobyte sehr gering, müssen aber für eine interaktive Oberfläche

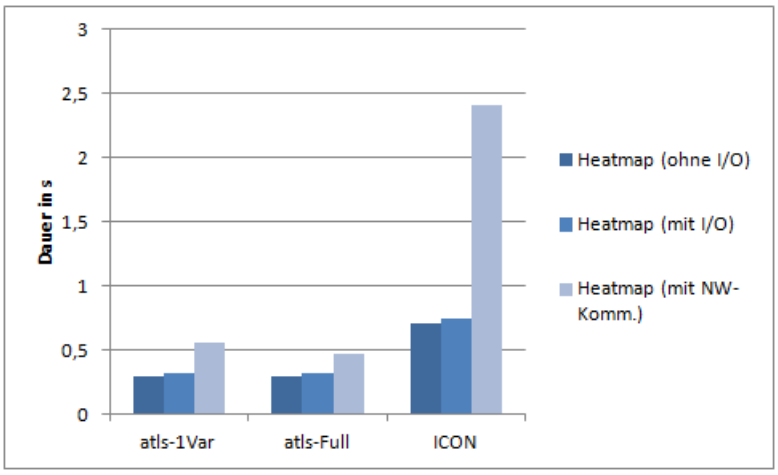

Abbildung 5.8.: Vergleich von Experiment 1, 2 und 3 (Heatmap)

schnell aus den NetCDF-Dateien gelesen werden.

	atls-1Var	**atls-Full**	**ICON**
Anzahl Anfragen	39	111	224
Dauer der Anfragen (aufsummiert)	1,3 s	8,37 s	23,7 s
Gesamtgröße der Antworten (JSON)	68,6 kB	154 kB	363 kB
CPU-Auslastung (Server)	12% - 44%	12% - 60%	48% - 60%
RAM-Bedarf (Server)	1,12 MB	1,94 MB	11,6 MB

Tabelle 5.3.: Messergebnisse für Auswahl des Basis-Datasets

Die Messergebnisse sind in Tabelle 5.3 dargestellt. Es ist deutlich erkennbar, dass die aufsummierte Dauer der Anfragen mit der Anzahl der Anfragen zusammenhängt, ähnliches lässt sich auch für die Gesamtgröße der Antworten vermuten. Dabei muss jedoch beachtet werden, dass die einzelnen Anfragen nicht als gleichwertig betrachtet werden können. Die Auswahl des Basis-Datasets besteht sowohl aus umfangreichen Einzelabfragen, deren Antworten mehrere Attribute gleichzeitig enthalten können, während es auch kleine Anfragen gibt, die nur einzelne Werte abfragen. Es muss außerdem beachtet werden, dass einige Abfragen erst durchgeführt werden müssen, damit der Client Informationen zur nächsten anfallenden Abfrage erhält. Dementsprechend steigt für den Server die CPU-Auslastung aufgrund des ständigen Wechsels zwischen Netzwerkkommunikation, I/O-Zugriffen, Datenverarbeitung und -konvertierung mit der Anzahl der abzuarbeitenden Anfragen. Trotzdem wird *h5serv* auf nur einem CPU-Kern ausgeführt. Nur in

seltenen, für den Ansatz dieser Arbeit nicht praxisrelevanten Fällen ist es möglich, die Arbeit auf mehrere Kerne zu verteilen.

Die Abfrage der Daten zu einer 2D-Visualisierung fordert im Vergleich zum ersten Test Nutzdaten auf Basis der zuvor angeforderten Metadaten an. Diese Messung bezieht sich zeitlich nur auf die Dauer der Abfrage, die bereits zuvor fertig zusammengestellt wurde.

Tabelle 5.4 zeigt die Ergebnisse dieser Messung. Bei beiden `atls`-Dateien können die Daten ähnlich schnell abgefragt werden. Die minimalen Unterschiede in der Größe der Antwort sind durch unterschiedliche Testdaten zu erklären, da die Abfragen auf unterschiedlichen Datasets ausgeführt wurden. Die Messung der `ICON`-Datei weicht sowohl von den Messungen der `atls`-Dateien als auch von den Messungen zu der Datei aus den vorherigen Experimenten stark ab. Dies zeigt auch Abbildung 5.9. Die sehr hohe Bearbeitungszeit der Anfrage zur `ICON`-Datei für Experiment 3 ist eventuell erneut auf die bereits beobachtete Ungenauigkeit oder Leistungsinkonsistenz von `cURL` zurückzuführen. Trotzdem zeigt sich, dass einfache Visualisierungen effizient durchgeführt werden können, denn:

- Die zu visualisierende Datenmenge hat einen geringen Umfang und
- Die Daten können auch in größeren Dateien schnell ermittelt werden.

	atls-1Var	**atls-Full**	**ICON**
Dauer der Anfrage	2,306 s	3,523 s	0,347 s
Gesamtgröße der Antwort (JSON)	11,6 kB	13 kB	18,8 kB
Anzahl Datenpunkte	1096	1096	768
CPU-Auslastung (Server)	4% - 12%	4% - 8%	4%
RAM-Bedarf (Server)	5,66 MB	6,08 MB	N/A

Tabelle 5.4.: Messergebnisse für eine 2D-Visualisierung

Auch bei der Heatmap-Visualisierung werden Nutzdaten angefordert, diesmal werden jedoch eine zusätzliche Dimension beziehungsweise im Fall der `ICON`-Testdatei sogar zwei zusätzliche Dimensionen angefordert, was den Aufwand dieser Anfrage erhöht. Aufgrund der farblichen Darstellung von Werten in einer Heatmap steht diesem Visualisierungstyp im Vergleich zum Linien- oder Balkendiagramm eine zusätzliche Dimension zur Verfügung. Die Anzahl der benötigten Datenpunkte steigt dabei in Tests mit der `atls-Full`-Datei um den Faktor 105 bis 480 auf bis zu 115.000 Datenpunkte, die `ICON`-Datei benötigt bei der Visualisierung der gleichen Dimensionen (Breiten- und Längengrad) mit höherer

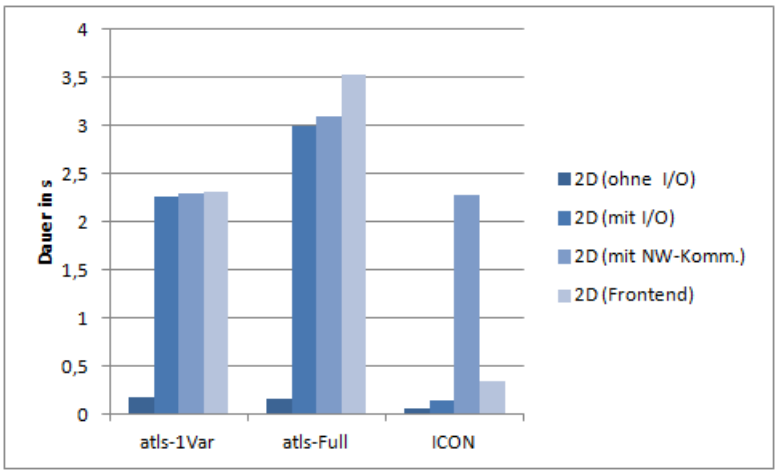

Abbildung 5.9.: Vergleich von Experiment 1, 2, 3 und 4 (2D-Visualisierung)

Auflösung knapp 295.000 Datenpunkte.

Die Messergebnisse zum Test zur Heatmap-Visualisierung sind in Tabelle 5.5 abgebildet. Dazu zeigt Abbildung 5.10 den direkten Vergleich der Heatmap-Visualisierung zu den vorherigen Experimenten. Es zeigt sich vor allem für die `ICON`-Datei, dass bereits die Datenmenge von knapp 5MB ausreicht, um die Bearbeitungszeit im Vergleich zu den vorherigen Experimenten deutlich zu erhöhen, da der Browser die Datenmenge zunächst verarbeiten muss. Im Vergleich zur 2D-Visualisierung ist die Anzahl der angeforderten Datenpunkte und dementsprechend auch die Größe der versendeten Daten aufgrund der zusätzlichen Dimension deutlich gestiegen. Die CPU-Auslastung bleibt wie beim zweiten Test ähnlich gering. Die `ICON`-Testdatei benötigt im Vergleich zu den beiden kleineren Dateien mit knapp 15 MB wesentlich mehr Arbeitsspeicher. Auch dies ist auf die Anordnung der Daten innerhalb der Datei zurückzuführen. Hierbei erhöht sich aufgrund einer vierten Dimension (Zeit, Level, Breitengrad, Längengrad) die Komplexität der Anfrage. Im Gegensatz zu den beiden anderen Dateien steigt das Verhältnis der Antwortgrößen stärker als das Verhältnis der angeforderten Datenpunkte an. Bei einer Zunahme der Datenpunkte um den Faktor 2,5 ist die Größe der Antwort um den Faktor 6,4 größer. Zurückzuführen ist dies unter anderem wie bereits beschrieben auf den verwendeten Datentyp der Datasets.

Für die Heatmap-Visualisierung wurde außerdem der Anteil der Netzwerkkommunikation an der gesamten Bearbeitungszeit ermittelt. Kurze Tests ergaben dabei einen durchschnittlichen Durchsatz von circa 1,1 MB/s. Von diesem Wert ausgehend ergeben sich für die Netzwerkkommunkation hohe Anteile an der Gesamtzeit von circa 25% bis 60%. Einschränkungen in der Leistungsfähigkeit des Konzeptes sind also durch das Netzwerk und weniger durch die Software-Komponenten zu erwarten.

In Tabelle 5.5 ist vor allem bei der `atls-1Var`-Datei der Unterschied des Bedarfs an

	atls-1Var	atls-Full	ICON
Dauer der Anfrage	2,495 s	1,189 s	7,033 s
Anteil Netzwerkkommunikation	0,60 s	0,66 s	4,264 s
Größe der Antwort (JSON)	662,4 kB	735,8 kB	4,69 MB
Anzahl Datenpunkte	115.680	115.680	294.912
CPU-Auslastung (Server)	12% - 16%	8% - 16%	64% - 96%
RAM-Bedarf (Server)	0,98 MB	0,07 MB	14,6 MB

Tabelle 5.5.: Messergebnisse für eine Heatmap-Visualisierung

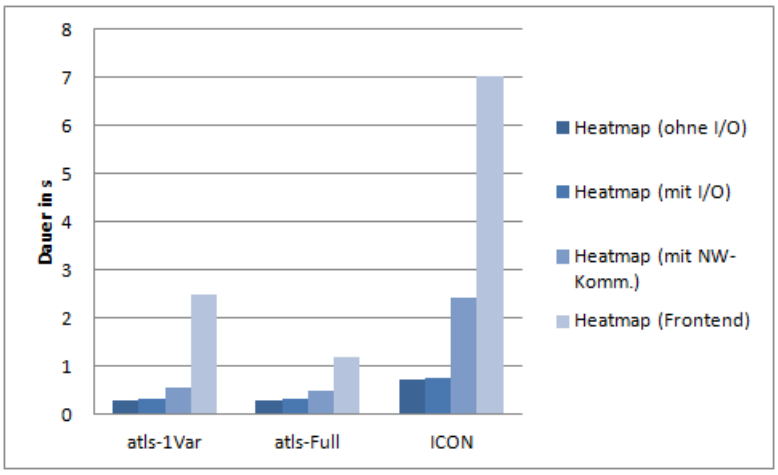

Abbildung 5.10.: Vergleich von Experiment 1, 2, 3 und 4 (Heatmap)

Arbeitsspeicher im Vergleich zu Tabelle 5.4 interessant. Liegt das Verhältnis bei dieser Messung von Nutzdaten zu RAM-Bedarf bei ungefähr 1:514, sinkt dieses Verhältnis bei der Messung zur Heatmap-Visualisierung (Tabelle 5.5) auf circa 1:1,5. Der eigentliche Bedarf sinkt mit knapp unter einem Megabyte sogar auf fast $\frac{1}{6}$ des Bedarfs der 2D-Visualisierung.

Die Vermutung liegt nahe, dass dies mit der Anordnung der angeforderten Daten zusammenhängt. Wie in Abschnitt 2.1.1 erläutert wurde, werden die Daten in NetCDF-Dateien in n-dimensionalen Arrays abgelegt. Die Web-Oberfläche der Heatmap-Visualisierung berücksichtigt zur Zeit noch keine Auswahl der Dimensionen. Somit werden alle Werte für Längen- und Breitengrad verwendet, für die Zeitdimension wird jedoch ein fester Wert ausgewählt. Der Speicheraufwand ist bei der in Abbildung 5.11 skizzierten Anordnung der Daten gering, wenn aus dem übergeordneten Array für `time` für den ausgewählten Index einfach nur alle Daten ausgelesen werden müssen. Alle benötigten Werte für den

Zeitpunkt i wären damit über `values[i]` sofort auslesbar.

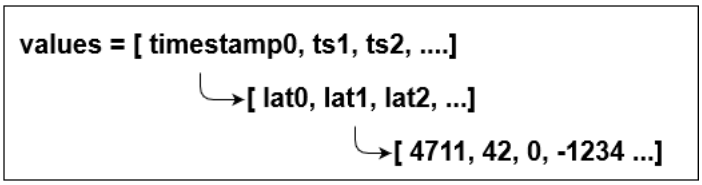

Abbildung 5.11.: Anordnung der Daten (skizziert)

Das Auslesen aller Daten für einen bestimmten Zeitpunkt ist mit dieser Anordnung dementsprechend nicht speicherintensiv. Im Gegensatz dazu ist vermutlich eine Anfrage, bei der alle Daten für einen festen Breitengrad-Wert angefordert werden, wesentlich aufwendiger zu verwalten. Tabelle 5.6 zeigt den Vergleich zwischen der Standard-Anfrage (fester Wert aus der Zeitdimension) und einer künstlich erzeugten Anfrage (fester Wert aus der Breitengrad-Dimension). Die Vermutung bestätigt sich: Das Verhältnis von Nutzdaten zu Speicherbedarf steigt auf circa 1:11,5.

Abfrage für `atls-1Var`	**Standard**	**Dimensionstausch**
Dauer der Anfrage	0,108 s - 0,858 s	5,024 s
Gesamtgröße der Antwort	662,3 kB	3,54 MB
Anzahl Anfragen	1	1
Anzahl Datenpunkte	115.680	526.080
CPU-Auslastung (Server)	12% - 16%	24% - 32%
RAM-Bedarf (Server)	0,98 MB	40,77 MB

Tabelle 5.6.: Messergebnisse bei künstlichem Dimensionstausch

Experiment 5: OpenCPU

Bei diesem Experiment wird eine Funktion zur Dimensionsreduktion auf ein Dataset ausgeführt, welches im vorherigen Schritt angefordert wurde. Dieser Test hat im Vergleich zu den vorherigen Tests einen höheren Aufwand, da der Arbeitsschritt auf den Daten eines gesamten Datasets arbeitet. Er repräsentiert eine CDO-Ausführung, die mit der Ausführung aus Listing 5.1 vergleichbar ist.

Die Pipeline, die nur aus Funktionen der implementierten `R`-Bibliothek besteht, konnte nicht innerhalb von 200 Sekunden ausgeführt werden, wodurch OpenCPU die Ausführung

der Anfrage mit einem *Timeout* automatisch beendete. Da dieses Ergebnis für den Nutzer nicht zufriedenstellend ist, empfiehlt es sich zu untersuchen, ob das Problem bei OpenCPU oder bei der eigenen Bibliothek liegt. Dazu wird die gleiche Pipeline als Funktion ohne OpenCPU in der Konsole ausgeführt. Die Ergebnisse zeigt Tabelle 5.7.

	OpenCPU + R	**R (Konsole)**
Bearbeitungszeit *h5serv*	55,9 s	47,7 s
Bearbeitungszeit `R`	$>$ 144,1 s	180,8 s
Bearbeitungszeit insgesamt	$>$ 200 s (`Timeout`)	228,5 s

Tabelle 5.7.: Messergebnisse *OpenCPU* im Vergleich zur `R`-Konsole

Auch wenn es zwischen den Ergebnissen der *OpenCPU*-Messung und der `R`-Konsolen-Messung leichte Abweichungen, beispielsweise im Zeitbedarf der Datenanfrage von *h5serv*, gibt, zeigt sich als Kernaussage der Messung eindeutig, dass die Implementation eigener Funktionalität keine akzeptable Bearbeitungszeit für komplexere Anfragen erzielt. Für beide Messungen ergibt sich für den Nutzer eine nicht zufriedenstellende Wartezeit von über drei Minuten.

Es wurde daher ein alternativer Ansatz implementiert, der die Bereitstellung von CDO-Funktionalität über *OpenCPU* ermöglichen soll. Die Ausführung erfolgt gemäß der genannten Konstellation 7 aus Abschnitt 3.1.3. Dabei werden Funktionen von CDO per `R`-*System Call* von *OpenCPU* aus aufgerufen. Eine wichtige Kenngröße ist hierbei vor allem der Overhead von OpenCPU im Vergleich zur "direkten" Ausführung von CDO als Konsolenanwendung.

Die gewählte CDO-Anfrage wird in Listing 5.1 gezeigt und ermittelt den Durchschnittswert eines Datasets pro Monat (`monmean`, *monthly mean*). Einbezogen werden für jeden Zeitpunkt (einmal täglich) die verfügbaren Werte als Durchschnitt für Längen- (`zonavg`, *zonal average*) und Breitengrad (`meravg`, *meridional average*).

```
1 cdo -L selvar,t2m -zonavg -meravg -monmean <ifile> <ofile>
```

Listing 5.1: CDO-Ausführung: Ermittlung des (weltweiten) monatlichen Mittels

Tabelle 5.8 zeigt die Messergebnisse der Ausführung über *OpenCPU* im Vergleich zur Ausführung von CDO als Konsolenanwendung.
Bei der Ausführung der CDO-Anweisungen muss beachtet werden, dass bei einem *System Call* von *OpenCPU* ein Overhead anfällt. Im Experiment belief sich dieser auf circa 4,8 Sekunden. Die Antwort von CDO umfasst nur wenige Zeilen Text, die in JSON formatiert an den Nutzer zurückgegeben werden. Daher fällt zusätzlich Netzwerkkommunikation an, diese ist jedoch im Vergleich zum *OpenCPU*-Overhead zu vernachlässigen. Die CDO-

	OpenCPU + CDO	**CDO (Konsole)**
Zeitanteil CDO	13,84 s	13,92 s
Zeitanteil OpenCPU + NW	4,77 s	0 s
Insgesamt	**18,61 s**	**13,92 s**

Tabelle 5.8.: Messergebnisse zur Ausführung von CDO

Ausführung selbst zeigt im Vergleich der beiden Ausführungsmethoden nur minimale Unterschiede von wenigen Millisekunden. Bei den Messungen sind durch *OpenCPU* außerdem nur wenige Ressourcen beansprucht worden, da der *OpenCPU*-Prozess in der Zeit der CDO-Ausführung nur auf den Erhalt der Antwort von CDO wartet.

Die Anbindung von CDO sowie der Aufruf dieser Anwendung durch *OpenCPU* bieten daher eine sehr gute Alternative zur Implementation einer eigenen `R`-Bibliothek. Abgesehen von wenigen Sekunden Overhead durch *OpenCPU* und Netzwerkkommunikation erzielt der *System Call*-Ansatz ähnlich gute Ergebnisse wie die direkte Ausführung von CDO als Konsolenanwendung. Zusätzlich steht damit dem Nutzer die gesamte Funktionalität von CDO zur Verfügung.

5.4. Fazit

Die Messergebnisse und Analysen aus Abschnitt 5.3 haben aussagekräftige Erkenntnisse in Bezug auf die Nutzung der vorgestellten Komponenten ergeben.

Für das **Frontend** hat sich gezeigt, dass clientseitige Visualisierungen eine aussagekräftige Menge an Datenpunkten verarbeiten können, ohne dass der Nutzer lange Wartezeiten hat. Es ist dennoch stets darauf zu achten, dass das Frontend mit einem möglichst kleinen Teil der Daten in Berührung kommt. Größere Datenmengen im kleinen zweistelligen Megabyte-Bereich können auch moderne Browser schon in Schwierigkeiten bringen. Im Bereich der Metadaten-Verarbeitung zeichnet sich das Frontend um *AngularJS* durch schnelle Reaktionszeiten aus. Auch hier sollte man bei der Entwicklung darauf achten, dass Anfragen möglichst klein gehalten werden, ohne dass zu viele unnötige Einzelabfragen gestellt werden.

Das **Backend** in Form von *OpenCPU* zeigt große Schwächen bei der Ausführung von Datenverarbeitungsfunktionen aus einer selbst implementierten `R`-Bibliothek. Mit Ausführungszeiten von mehreren Minuten für verhältnismäßig einfache Funktionen stellt sich diese Nutzung als nicht praktikabel dar. Die Möglichkeit, für *OpenCPU* eine Anbindung an CDO zu implementieren, stellt hingegen eine leistungsfähige Alternative dar, die mehrere Vorteile bietet:

- Die CDO-Ausführung ist für beide Ausführungsmöglichkeiten ähnlich effizient,
- *OpenCPU* erzeugt nur einen geringen Overhead im Vergleich zur CDO-Ausführung als Konsolenanwendung,
- Es müssen bei der CDO-Ausführung keine Nutzdaten an den Webbrowser übertragen werden, da das Ergebnis als eigene NetCDF-Datei dem *h5serv*-Index hinzugefügt wird und
- Dem Nutzer steht die gesamte CDO-Funktionalität (mehrere 100 Funktionen) zur Verfügung.

h5serv zeigte als **Datenquelle**, dass es die angeforderten Daten schnell bereitstellen konnte. Das Anfrageschema ist einfach und verständlich gehalten und kann vom Frontend durch eine geschickt aufgebaute Weboberfläche gut zugänglich gemacht werden. Gerade im Bereich der Metadaten ist *h5serv* eine gute Wahl, auch Abfragen zu kleineren Datensätzen funktionieren schnell und zuverlässig. Größere Datenmengen können ebenfalls abgefragt werden, jedoch fehlen hier zur Zeit die Komponenten, die diese Datenmengen effizient und schnell bearbeiten können. Die CPU-Auslastung des Servers hielt sich bei den praxisnahen Beispielabfragen in Grenzen, auch die RAM-Nutzung lag oftmals bei nur wenigen Megabyte. *h5serv* konnte je nach Typ der Anfrage durch die Nutzung von `tmpfs` diese um bis zu 18 mal schneller beantworten, teilweise war allerdings auch kaum ein Zeitersparnis messbar. Die Anfragen konnten jedoch auch mit I/O-Zugriffen in für den Benutzer zufriedenstellender Zeit beantwortet werden.

Zusammenfassung

Das Kapitel zeigte einige Beispielmessungen, die die Leistungsfähigkeit der unterschiedlichen verwendeten Komponenten analysieren sollte. Für das Frontend und h5serv als Datenquelle ergaben sich keine größeren Leistungsinkonsistenzen, die ein Austausch der Komponeten notwendig gemacht hätten. Für eine nicht zufriedenstellende Implementation einer eigenen `R`-Bibliothek für Datenverarbeitungsfunktionalität konnte durch die Anbindung von CDO eine effizient arbeitende Alternative gefunden werden. Mit zusätzlichen Anpassungen, die die Ergebnisse dieses Kapitels berücksichtigen, stellt der in dieser Arbeit vorgestellte Ansatz eine gute Möglichkeit dar, Voranalysen auf NetCDF-Dateien über den Webbrowser steuern zu können.

6. Zusammenfassung

Diese Kapitel fasst noch einmal die Ergebnisse dieser Arbeit zusammen. Abschließend wird in Abschnitt 6.2 ein Ausblick auf Erweiterungsmöglichkeiten und zukünftige Schwerpunkte der vorgestellten Webanwendung gegeben.

6.1. Zusammenfassung

Nachdem in Kapitel 1 die Untersuchung des beschriebenen Konzeptes auf Basis von *OpenCPU* und *h5serv* als Motivation für die Arbeit genannt und anschließend die Ziele der Arbeit beschrieben wurden, befasste sich Kapitel 2 mit eingesetzten Technologien sowie verwandten Software-Systemen. Kapitel 3 stellte den Aufbau der Webanwendung vor. Dabei wurden detailliert die Komponenten von Frontend, Backend und Datenquelle erläutert, um darauf aufbauend mögliche Konstellationen der Komponenten zu analysieren, die für die unterschiedlichen Aufgabengebiete der Webanwendung verwendet werden können. Zur Verdeutlichung wurden häufig vorkommende Abläufe innerhalb der Anwendung gezeigt. Kapitel 4 erläuterte mit der Asynchronität von HTTP-Anfragen einen der Kernaspekte der Implementation, bevor die einzelnen Komponenten des Frontends im Detail beschrieben wurden. Mit den Informationen zum Aufbau und zur Funktionsweise der Anwendung konnten in Kapitel 5 unterschiedliche Tests formuliert und deren Ergebnisse analysiert werden. Dadurch konnten die verwendeten Komponenten auf ihre Nutzbarkeit für das vorgestellte Konzept untersucht werden. Es zeigte sich, dass *OpenCPU* und *h5serv* eine solide Basis bieten, um in Kombination mit einem modernen Frontend dem Benutzer eine Plattform zur Verfügung zu stellen, mit der er Voranalysen zu NetCDF-Dateien bequem über den Webbrowser steuern kann.

Neben der Möglichkeit, einzelne (Sub-)Datasets aus NetCDF-Dateien visuell im Browser darzustellen, bietet `ReDaVis` dem Nutzer außerdem an, zunächst mehrere Funktionen zur Datenverarbeitung auf angeforderte Daten als Pipeline auszuführen. Dabei ermöglicht der modulare Aufbau des Frontends, den Umfang an Visualisierungs- und Datenverarbeitungsbibliotheken zu erweitern. Durch die Anbindung von CDO steht dem Nutzer bereits jetzt ein großer Umfang an ausführbaren Funktionen zur Verfügung.

6.2. Ausblick

Im Laufe der Entwicklung der vorgestellten Webanwendung wurden zahlreiche Ideen entwickelt, inwiefern der Funktionsumfang noch erweitert werden kann. Für die Bearbeitungszeit musste allerdings ein realistischer Umfang angestrebt werden, sodass einige

Ideen erst einmal zurückgestellt wurden. Dabei war es wichtig darauf zu achten, inwiefern der Nutzen einer zusätzlichen Funktionalität den Mehraufwand für die Entwicklung rechtfertigte.

Die Anwendung bietet somit zunächst eine solide Funktionsgrundlage, die die wesentlichen Untersuchungen auf Machbarkeit ermöglichen. In zukünftigen Versionen kann dann der Umfang an Funktionalität durch folgende Ideen erweitert werden:

Umfangreichere Visualisierungsmöglichkeiten

Die momentan aktuelle Version der Anwendung bietet in Bezug auf die Visualisierung zwei Sub-Services mit insgesamt drei Visualisierungsmöglichkeiten (Linien- und Balkendiagramm sowie eine Heatmap) an. Damit konnte die grundliegende Idee hinter dem *VisualizationService* getestet werden. Zukünftig könnten weitere JavaScript-Bibliotheken eingebunden werden, die über eigene Sub-Service zusätzliche interaktive Visualisierungsmethoden anbieten.

Bereitstellung einer Benutzeroberfläche für die CDO-Nutzung

Da die CDO-Anbindung erst spät als Alternative zur eigenen `R`-Bibliothek implementiert wurde, gibt es mit der aktuellen Version von `ReDaVis` nur eine einfache Eingabeseite, über die CDO-Befehle ähnlich wie bei der Konsolenanwendung ausgeführt werden. Zukünftig sollte der Fokus vor allem auf einer verbesserten Anbindung der CDO-Funktionalität liegen. Dafür sollte die Benutzeroberfläche, die zur Zeit schon für die eigene `R`-Bibliothek implementiert ist, entsprechend für die unterschiedlichen CDO-Funktionen angepasst werden, sodass dann auch über wenige und einfache Benutzereingaben eine CDO-Pipeline aufgebaut und ausgeführt werden kann.

Erweiterung des Pipeline-Kerns

Eine potentiell effizientere Nutzung von CDO durch *OpenCPU* oder gegebenenfalls auch durch andere Backendkomponenten würde die Erweiterung des Pipeline-Aufbaus im Frontend fördern. Außerdem könnte der Aufbau der Pipeline über eine umfangreichere Benutzeroberfläche bequemer gestaltet werden. Pipeline-Schritte könnten dann beispielsweise per Drag&Drop angeordnet werden.

Ansprechenderes Frontend

Im Allgemeinen kann die Web-Oberfläche optisch ansprechender gestaltet werden. Gerade durch die Nutzung von Frameworks wie *Bootstrap* ergeben sich zahlreiche Möglichkeiten, eine ansprechende, interaktive Benutzeroberfläche zu entwickeln. Auch wenn der Fokus der Arbeit auf der Funktionalität lag, zeigte sich an einigen Stellen bereits das Potential der verwendeten Frameworks.

User-Management und langfristige Sitzungen

Zur Zeit arbeitet die Webanwendung zustandslos. Solange man die Website nicht verlässt, werden Daten-Zusammenhänge und vermeintlich "gesicherte" Daten wie *Snapshots* zwar aufbewahrt, das erneute Aufrufen der Website wird jedoch als neue Session erkannt und bereits durchgeführte Anfragen und ähnliches sind dann nicht mehr verfügbar. Komplexere Funktionalität in Bezug auf langfristige, wiederverwendbare Sessions oder auch Aspekte wie User-Management erfordert die aufwendige Entwicklung einer zusätzlichen Backend-Komponente, die sämtliche zu speichernde Daten in einer Datenbank verwalten muss.

Vereinfachter Austausch von Ergebnissen

Der Austausch von Ergebnissen kann zur Zeit schon leicht über die *Session Keys* von *OpenCPU* erfolgen. Dieser Austausch kann für zukünftige Versionen noch weiterhin verbessert werden. Durchgeführte Anfragen könnten über kleinere Dialog ähnlich wie die Snapshots langfristig gespeichert werden. Für die Speicherung der auszutauschenden Daten wäre jedoch die Entwicklung einer in Abschnitt 6.2 beschriebenen Komponente nötig.

Anhänge

A. Installation der Backend-Komponenten

Die Installation der verwendeten Komponenten erfolgte nach lokalem Test auf dem in Abschnitt 5.1 vorgestellten Cluster des DKRZ.

A.1. Installation von OpenCPU

Die Installation von *OpenCPU* erfolgte nach Anleitung[1]. Danach sollten für diesen speziellen Anwendungsfall noch einige Konfigurationen angepasst werden. Die Installation sieht wie folgt aus:

```
1 # Requires Ubuntu 14.04 (Trusty) or 16.04 (Xenial)
2 sudo add-apt-repository -y ppa:opencpu/opencpu-1.6
3 sudo apt-get update
4 sudo apt-get upgrade
5
6 # Installs OpenCPU server
7 sudo apt-get install -y opencpu
```

Listing A.1: Installation von OpenCPU

OpenCPU wird als Service installiert und wird damit bei Systemstart bereits initialisiert. Nach abgeschlossener Installation ist die Testseite von *OpenCPU* über den Webbrowser unter `http://<server_domain>/ocpu` erreichbar. Alle folgenden Aufrufe der Komponente werden ebenfalls über diese Adresse ausgeführt.

Je nach verfügbarem System sollten einige Optionen in der Konfigurationsdatei unter `/etc/opencpu/server.conf` angepasst werden. Bei Verwendung einer eigenen Backend-Bibliothek sollte beispielsweise die maximale Ausführungszeit von `R` (`timelimit.get` beziehungsweise `timelimit.post`) angepasst werden. Es ist außerdem vor allem bei größeren Datensätzen nötig, die Arbeitsspeicher-Begrenzung von 1 GB pro Anfrage anzupassen. Diese Begrenzung wird von dem `RAppArmor`-Package verwaltet ([Oom13b], S. 15). Für die CDO-Anbindung sind weitere Konfigurationen von `RAppArmor` nötig, die es dem CDO-Systemaufruf ermöglichen, auf das `/data/`-Verzeichnis von *h5serv* zugreifen zu können.

[1] Hinweise zur Installation unter verschiedenen Betriebssystemen sind unter `https://www.opencpu.org/download.html` aufgeführt.

A.2. Installation von h5serv

Die Installation von *h5serv* ist einfach gehalten[2], da man nur die Git-Projekte zu *h5serv* und *hdf5-json* kopieren muss:

```
1 git clone https://github.com/HDFGroup/hdf5-json.git
2 cd hdf5-json
3 python setup.py install
4
5 git clone https://github.com/HDFGroup/h5serv.git
```

Listing A.2: Installation von h5serv

Für die Verwendung von *h5serv* müssen folgende Abhängigkeiten bereitgestellt werden:

- NumPy 1.10.4 oder aktueller
- h5py 2.5 oder aktueller
- tornado 4.0.2 oder aktueller
- watchdog 0.8.3 oder aktueller
- requests 2.3 oder aktueller

Dafür wurde eine virtuelle `Python`-Umgebung mithilfe von *VirtualEnv*[3] eingerichtet, die diese Abhängigkeiten verwaltet.

h5serv muss zur Zeit manuell gestartet werden. Dafür wird ein Shell-Skript aufgerufen, das zunächst erst die virtuelle Python-Umgebung lädt und dann *h5serv* startet. Anfragen können dann über den Standardport `5000` gestellt werden, dieser ist aber bei Bedarf auch konfigurierbar.

[2] `https://github.com/HDFGroup/h5serv#introduction`
[3] `http://docs.python-guide.org/en/latest/dev/virtualenvs/`

Literaturverzeichnis

[CW11] Iskren Ivov Chernev and Tim Wood. Moment.js 2.13.0 - Parse, validate, manipulate, and display dates in JavaScript. http://momentjs.com/, 2011.

[dlB06] Jeff de la Beaujardiere, editor. *OpenGIS Web Map Service Implementation Specification, Version 1.3.0.* Open Geospatial Consortium Inc., 2006.

[fM12] Max-Planck-Institut für Meteorologie. CDO Tutorial. https://code.zmaw.de/projects/cdo/wiki/Tutorial#Combining-operators, 2012.

[GPS+07] James Gallagher, Nathan Potter, Tom Sgouros, Steve Hankin, and Glenn Flierl. The Data Access Protocl - DAP 2.0. http://www.opendap.org/pdf/ESE-RFC-004v1.1.pdf, 2007.

[gro16] The HDF group. High Level Introduction to HDF5. https://www.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf, 2016.

[Heb14] Gerd Heber. RESTful HDF5 - Interface Specification - Version 0.1. https://www.hdfgroup.org/pubs/papers/RESTful_HDF5.pdf, 2014.

[MW16] Pavel Michna and Milton Woods. Package 'RNetCDF' - Interface to NetCDF Datasets. https://cran.r-project.org/web/packages/RNetCDF/, 2016.

[Oom13a] Jeroen Ooms. OpenCPU - Producing and Reproducing Results. https://www.opencpu.org/, 2013.

[Oom13b] Jeroen Ooms. The RAppArmor Package: Enforcing Security Policies in R Using Dynamic Sandboxing on Linux. *Journal of Statistical Software*, 55(1):1–34, 2013.

[Oom14] Jeroen Ooms. The OpenCPU System: Towards a Universal Interface for Scientific Computing through Separation of Concerns. http://arxiv.org/pdf/1406.4806v1.pdf, 2014.

[Ope07] Open Geospatial Consortium. *Web Processing Service Standard˜OGC˜05-007r7*, volume 1.0.0. Open Geospatial Consortium, 2007.

[OT10] Mark Otto and Jacob Thronton. Bootstrap - The world's most popular mobile-first and responsive front-end framework. http://getbootstrap.com/getting-started/, 2010.

[RDE+16] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, Ed Hartnett, Dennis Heimbigner, and Ward Fisher. Netcdf 4.4.0 faq. `http://www.unidata.ucar.edu/software/netcdf/docs/faq.html#formatsdatamodelssoftwarereleases`, 2016.

[Sch16] Uwe Schulzweida. CDO User's Guide. `https://code.zmaw.de/projects/cdo/embedded/cdo.pdf`, 2016.

[TAT] The AngularUI Team. AngularUI for AngularJS. `https://angular-ui.github.io/bootstrap/`.

[TB14] Philipp Tarasiewicz and Robin Böhm. *AngularJS: Eine praktische Einführung in das JavaScript-Framework*. Dpunkt.Verlag GmbH, 2014.

[The16] The HDF Group. Hierarchical Data Format, version 5, 1997-2016. http://www.hdfgroup.org/HDF5/.

Abbildungsverzeichnis

Listingübersicht

Tabellenverzeichnis

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Master-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 21.07.2016 .....