Westfälische Wilhelms-Universität Münster
Institut für Geoinformatik

Bachelorarbeit
im Fach Geoinformatik

Rich Data Interfaces for Copernicus Data

Themensteller: Prof. Dr. Albert Remke
Betreuer: Dr. Christian Knoth, Dipl.-Geoinf. Matthes Rieke
Ausgabetermin: tbd.
Abgabetermin: tbd.

Vorgelegt von: Alexander Nicolas Pilz
Geboren: 06.12.1995
Telefonnummer: 0176 96982246
E-Mail-Adresse: apilz@uni-muenster.de
Matrikelnummer: 512 269
Studiengang: Bachelor Geoinformatik
Fachsemester: 6. Semester

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

OGC Open Geospatial Consotium

SAR Synthetic Apeture Radar

ESA European Space Agency

GMES Global Monitoring for Environmental Security

CAMS Copernicus Atmosphere Monitoring Service

CMEMS Copernicus Marine Environment Monitoring Service

CLMS Copernicus Land Monitoring Service

EMS Emergency Management Service

C3S Climate Change Service

SM Stripmap Mode

SLC Single Look Complex

GRD Ground Rage Detected

OSW Ocean Swell Spectra

OWI Ocean Wind Field

RVL Radial Surface Velocity

NDSI Normalized Difference Sigma-Naught Index

DIAS Data and Information Access Services

JSON Java Script Object Notation

URL Uniform Resource Locator

XML Extensible Markup Language

WSGI Web Server Gateway Interface

REST Representational State Transfer

1 Einleitung

1.1 Motivation

Geodaten helfen weltweit bei der Analyse unterschiedlichster Probleme. Um diese Analysen möglichst schnell und präzise durchführen zu können müssen Geodaten schnell und in hoher Qualität abrufbar sein. Zwar lassen sich zahlreiche Geodaten nach Bedarf aus dem Internet beziehen, manche Geodaten bedürfen allerdings zeit- und und rechenintensiven Vorverarbeitungen welche teilweise fundierte Kenntnisse aus dem Bereich der Geoinformatik voraussetzen. Zur Lösung dieses Problems könnten sogenannte Rich Data Interfaces dienen. Diese würden Nutzer in die Lage versetzen Geodaten abzurufen welche bereits auf einen speziellen Anwendungsfall zugeschnitten sind.

1.2 Ziele

Ziel dieser Arbeit ist die Implementierung und Evaluation eines leichtgewichtigen, OGC API - Processes - Part: 1 Core Standard konformen Application Programming Interface. Die Implementierung soll dabei die Eigenschaften eines Rich Data Interface für Copernicus-Daten aufweisen. Dieses soll Rohdaten der Sentinel-1 Mission auf Nutzeranfrage hin beschaffen, vorverarbeiten und als Endprodukt Daten liefern welche sich für das Überschwemmungsmonitoring eignen. Es soll untersucht werden inwieweit sich der OGC API - Processes - Part: 1 Core Standard dazu eignet Rich Data Interfaces für Copernicus Daten zu entwerfen.

1.3 Aufbau

2 Grundlagen

2.1 Radarfernerkundung

Bei der Radarfernerkundung werden vom Radarsystem in regelmäßigen Abständen elektromagnetische Signale ausgesandt. Nach dem Senden eines Signals (Chirp) folgt ein Zeitfenster, indem die Plattform auf Echos des ausgesandten Signals wartet. Trifft das ausgesandte Signal auf eine Oberfläche, zum Beispiel die Erdoberfläche, wird ein Bruchteil in Richtung Empfänger reflektiert und als Echo vom Fernerkundungssystem empfangen [1].
Die Radarfernerkundung gehört zu den aktiven Fernerkundungsmethoden da hier im Gegensatz zur optischen Fernerkundung nicht nur von Oberflächen reflektierte Strahlung von anderen Strahlungsquellen wie der Sonne aufgenommen wird, sondern das Fernerkundungssystem selbst als Strahlungsquelle dient. Messungen können daher tageszeitunabhängig erfolgen. Bildgebende Radarsysteme werden auf mobilen Plattformen montiert und blicken seitlich auf die zu beobachtende Oberfläche. Die Flugrichtung wird Azimut und die Blickrichtung als Slant Range bezeichnet [1] (Abbildung 1).
Die Eigenschaften des reflektierten Signals hängen sowohl von Parametern des Aufnahmesystems als von Parametern der reflektierenden Oberfläche ab. So werden in der Radarfernerkundung verschiedenen Frequenzbänder verwendet, welche sich in Frequenz und Wellenlänge unterscheiden. Da sich die Wechselwirkungen zwischen Signalen unterschiedlicher Frequenzbänder und den reflektierenden Oberflächen unterscheidet können so unterschiedliche Aspekte der beobachteten Oberflächen hervorgehoben werden. Dabei kommen in der Regel Wellenlängen von 0.75m bis 120m zum Einsatz (siehe Tabelle 1). Mit einer größeren Wellenlänge kann ein Medium auch tiefer durchdrungen werden. Außerdem werden Wolken, Dunst und Rauch durchdrungen was den zusätzlich Vorteil bietet wetterunabhängig Messungen durchführen zu können [2].

Tabelle 1: Gängige Frequenz-Bänder in der Radarfernerkundung [1]

Frequenzband	Ka	Ku	X	C	S	L	P
Frequenz (GHz)	40-25	17.6-12	12-7.5	7.5-3.75	3.75-2	2-1	0.5-0.25
Wellenlänge (cm)	0.75–1.2	1.7–2.5	2.5–4	4–8	8–15	15–30	60–120

Die Durchdringungstiefe hängt auch von der Dielektrizitätskonstante, also der Leitfähigkeit, ab. Ist diese groß, kommt es zu starken Reflektionen und die Durchdringungstiefe ist gering. Die Rauigkeit ist eine Eigenschaft der reflektierenden Oberfläche und hat großen Einfluss auf das reflektierte Signal. Ist diese im Verhältnis zur verwandten Wellenlänge gering so kommt es zu spiegelnden Reflektionen und nur ein geringer Anteil des kehrt zum Empfänger zurück. Je diffuser die Reflektion mit zunehmender Rauigkeit wird umso größer ist der Anteil des Signals welcher zum Empfänger zurückgeworfenen Signals. Doch auch die Form und Exposition der Oberfläche nimmt Einfluss auf das reflektierte Signal. So werden Flächen je nach Nei-

gung unterschiedlich stark bestrahlt. Ist eine dem System abgewandte Fläche steiler geneigt als der Depressionswinkel liegen Sie sogar im Radarschatten und werden gar nicht bestrahlt [2]. Zusätzlich ist die Polarisation der ausgesandten und empfangenen Signale bei der Messung ausschlaggebend. Sie können horizontal oder vertikal polarisiert sein. Dies führt zu vier möglichen Polarisationsmodi für das Senden und das Empfangen nämlich HH, VV, HV und VH. Auch die Polarisation sorgt für eine unterschiedliche Wiedergabe von beobachteten Objekten und kann somit verwendet werden, um bestimmte Aspekte hervorzuheben [2]. Die Auflösung entlang des Azimut unterscheidet sich von der Auflösung in Blickrichtung. Die Auflösung in Azimutrichtung wird von der Antennenlänge bestimmt da diese festlegt wie lange die Reflektionen eines Objektes empfangen werden. Die Antennenlänge kann bauartbedingt nicht beliebig gesteigert werden. Die Bauart der Antenne bestimmt auch den Abstrahlwinkel Θ_a und somit die Ausdehnung am Boden eines Impulses in Azimutrichtung. Diese nimmt mit zunehmender Entfernung zu, während die Auflösung abnimmt. Die Auflösung in Blickrichtung hängt von der Bandbreite ab welche sich aus der Sendefrequenz und der Signaldauer. Die Ausdehnung des beobachteten Gebietes in Blickrichtung hängt von der Laufzeit des ausgesandten Signales ab. Die Objekte werden abhängig von ihrer Entfernung zur Antenne verzerrt wiedergegeben da nahegelegene Objekte von der Wellenfront schneller durchlaufen werden. Dieser Unterschied zwischen Schrägdistanz und Bodendistanz lässt sich jedoch nahezu vollständig korrigieren [2]. Die bisher beschriebenen Systeme werden auch als Systeme mit realer Apertur bezeichnet und eignen sich nur für geringe Flughöhen da hier der Abstand zwischen Antenne und Oberfläche gering ist. Bei Radarsystemen mit einer synthetischen Apertur wird durch die Bewegung des Sensors in Azimutrichtung die wirksame Antennenlänge rechnerisch verlängert indem die reflektierten Signale eines beobachteten Objektes von verschiedenen Standpunkten und unterschiedlichen Zeitpunkten miteinander korreliert werden. So können hohe Azimutauflösungen erzielt werden. Solche Systeme eigenen sich auch für den Einsatz auf Satelliten [2].

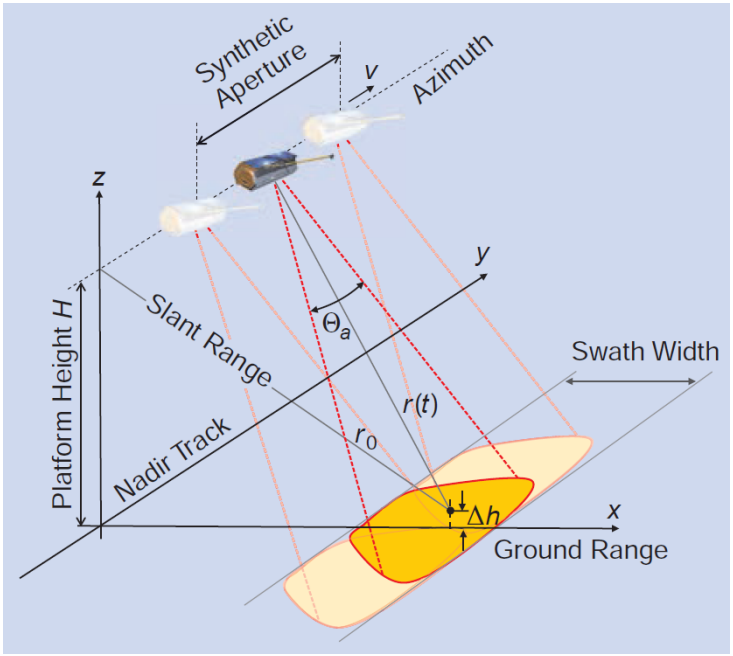

Abbildung 1: Prinzip eines SAR Fernerkundungssystems [1]

Solche Systeme können in unterschiedlichen Aufnahmeverfahren arbeiten. Das einfachste dieser Verfahren ist das Stripmap Verfahren bei dem nur ein Aufnahmestreifen kontinuierlich aufgenommen wird. Breitere Aufnahmestreifen können mit dem ScanSAR Verfahren erzielt werden. Dabei werden unter verschiedenen Depressionswinkeln, in Blickrichtung und zeitversetzt mehrere Subaufnahmestreifen erzeugt. Im Vergleich zum Stripmap Verfahren ist Auflösung jedoch geringer. Wird eine höhere Auflösung benötigt kann das Spotlight Verfahren zum Einsatz kommen, bei dem eine fixe Region über einen längeren Zeitraum hinweg beobachtet wird. Dies führt zu einer sehr langen wirksamen Antenne. Angepasste Verfahren oder Mischformen können je Beobachtungsszenario zum Einsatz kommen (siehe Abbildung 2) [1].

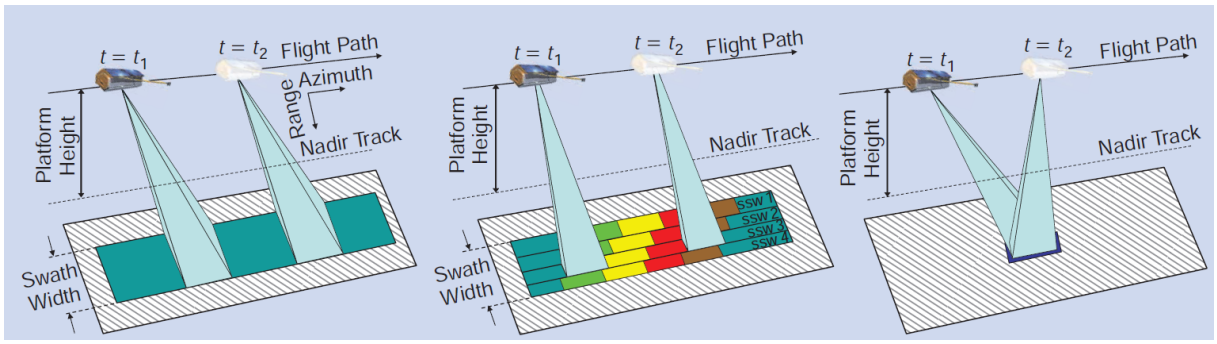

Abbildung 2: Aufnahmeverfahren SAR Systemen [1]

Im Gegensatz zu optischen Aufnahmeverfahren liefern die Rohdaten einer Befliegung mit Radarsensoren noch keine Bilddaten. Um Bilder zu erzeugen, bedarf es zunächst einer komplexen Verarbeitung der aus Amplitude und Phase bestehenden reflektierten Signale. Dabei werden

die Daten entlang des Azimuts und der Blickrichtung gefiltert. In der Regel repräsentieren die Pixelwerte eines aus Radardaten abgeleiteten Bildes die Reflektivität des korrespondierenden Bodenelements. Mittels Geocodierung kann das so entstandene Bild verortet werden. Zusätzlich können diverse, ebenfalls rechen- und zeitintensive, Kalibrierungen vorgenommen werden. Dazu gehören Verfahren welche Rauscheffekte minimieren, die geometrischen Eigenschaften verbessern oder die Interpretation der Bilder erleichtern [1].

2.2 Copernicus Programm

2.2.1 Ziele

Das Copernicus-Programm ging aus dem Global Monitoring for Environmental Security Programm (GMES) Programm hervor welches 1998 mit dem Ziel initiiert wurde um Europa zu ermöglichen eine führende Rolle bei der Lösung von weltweiten Problemen im Kontext Umwelt und Klima zu verschaffen. Teil dieser Bestrebungen ist der Aufbau eines leistungsfähigen Programms zur Erdbeobachtung. 2012 wurde das GMES-Programm zum Copernicus-Programm umbenannt [3]. Erklärte Ziele des Copernicus-Programmes ist das Überwachen der Erde um den Schutz der Umwelt sowie Bemühungen von Katastrophen- und Zivilschutzbehörden zu unterstützen. Gleichzeitig soll die Wirtschaft im Bereich Raumfahrt und der damit verbundenen Dienstleistungen unterstützt und Chancen für neue Unternehmungen geschaffen werden [7].

2.2.2 Aufbau

Das Copernicus-Programm besteht aus Weltraum, In-Situ- und Service-Komponente. Zur Weltraum-Komponente gehören die verschiedenen Satellitenmissionen sowie Bodenstationen welche für den Betrieb sowie die Steuerung und Kalibrierung der Satelliten sowie der Verarbeitung und Validierung der Daten verantwortlich sind [7].
Sentinel-1 Satelliten sind mit bildgebenden Radarsystemen ausgerüstet und beobachten wetter- und tageszeitunabhängig Land-, Wasser- und Eismassen, um unter andrem das Krisenmanagement zu unterstützen. Satelliten der Sentinel-2 Mission führen hochauflösende, multispektrale Kameras mit und liefern weltweit optische Fernerkundungsdaten.
Altimetrische und radiometrische Daten von Land- und Wasserflächen werden von der Sentinel-3 Satellitenmission gesammelt während spektrometrische Daten zur Überwachung der Luftqualität von Sentinel-4 und 5 Satelliten erfasst werden. Ozeanografische Daten sollen von den Sentinel-6 Satelliten geliefert werden [4].
Die In-Situ-Komponente sammelt Daten von See-, luft- und landbasierten Sensoren sowie geografische und geodätische Referenzdaten. Die harmonisierten Daten werden verwendet, um die Daten der Weltraum-Komponente zu verifizieren oder zu korrigieren. Gleichzeitig können räumliche oder thematische Lücken in der Datenabdeckung gefüllt werden [7] [5].

Zur Service-Komponente gehören unterschiedliche Dienste, welche jeweils auf Themengebiet abgestimmt sind und Daten in hoher Qualität bereitstellen. Der Copernicus Atmosphere Monitoring Service (CAMS) soll Informationen zur Luftqualität und der chemischen Zusammensetzung der Atmosphäre liefern. Daten bezüglich des Zustands und der Dynamik der Meere und deren Ökosysteme lassen sich über den Copernicus Marine Environment Monitoring Service (CMEMS) beziehen. Informationen zur Flächennutzung und Bodenbedeckung werden vom Copernicus Land Monitoring Service (CLMS) bereitgestellt. Um eine nachhaltige Klimapolitik planen und umsetzen zu können stellt der Copernicus Climate Change Service (C3S) aktuelle sowie historische Klimadaten bereit. Um den Zivilschutzbehörden schnelle Reaktionen auf Umweltkatastrophen zu ermöglichen, stellt der Emergency Management Service (EMS) entsprechende Fernerkundungsdaten breit. Ähnliche Daten können von europäischen Zoll- und Grenzschutzbehörden über den Copernicus Security Service bezogen werden [7] [5].

2.2.3 Sentinel 1

Die Sentinel-1 Satellitenmission liefert wetter- und tageszeitunabhängige Radardaten der Erdoberfläche. Die Mission besteht aus zwei Satelliten, Sentinel-1 A und B, sowie einer Bodenkomponente welche für Steuerung, Kalibrierung und Datenverarbeitung verantwortlich ist. Die Satelliten tragen als Hauptinstrument ein bildgebendes Radar mit synthetischer Apertur welches im C-Frequenzband arbeitet. Es stehen zwei Polarisationsmodi, Single (HH, VV) oder Dual (HH+HV, VV+VH), zur Verfügung [8]. Die Erfassung von Daten kann in vier Aufnahmemodi erfolgen welche sich in Auflösung, Streifenbreite und Anwendungsszenario unterscheiden (siehe Tabelle 2). Der Standardmodus ist der Stripmap Modus (SM) bei dem Aufnahmestreifen mit einer kontinuierlichen Folge von Signalen abgetastet wird [8]. Die Aufnahmemodi Interferometric Wide Swath Mode (IW) und Extra-Wide Swath Mode (EW) arbeiten im TOPSAR Verfahren mit drei beziehungsweise fünf Sub-Aufnahmestreifen um ein größeres Gebiet aber in geringerer Auflösung aufnehmen zu können. TOPSAR ist eine Abwandlung des ScanSAR Verfahrens bei dem die Antenne zusätzlich in Azimut-Richtung vor und zurück bewegt wird, um die radiometrische Qualität der resultierenden Bilder zu verbessern. Wenn der Wave Modus (WV) zu Einsatz kommt werden kleine, Vignetten genannte, Szenen im Stripmap Verfahren aufgenommen. Sie werden in regelmäßigen Abständen und wechselnden Depressionwinkeln aufgenommen (siehe Abbildung 3) [1] [8].

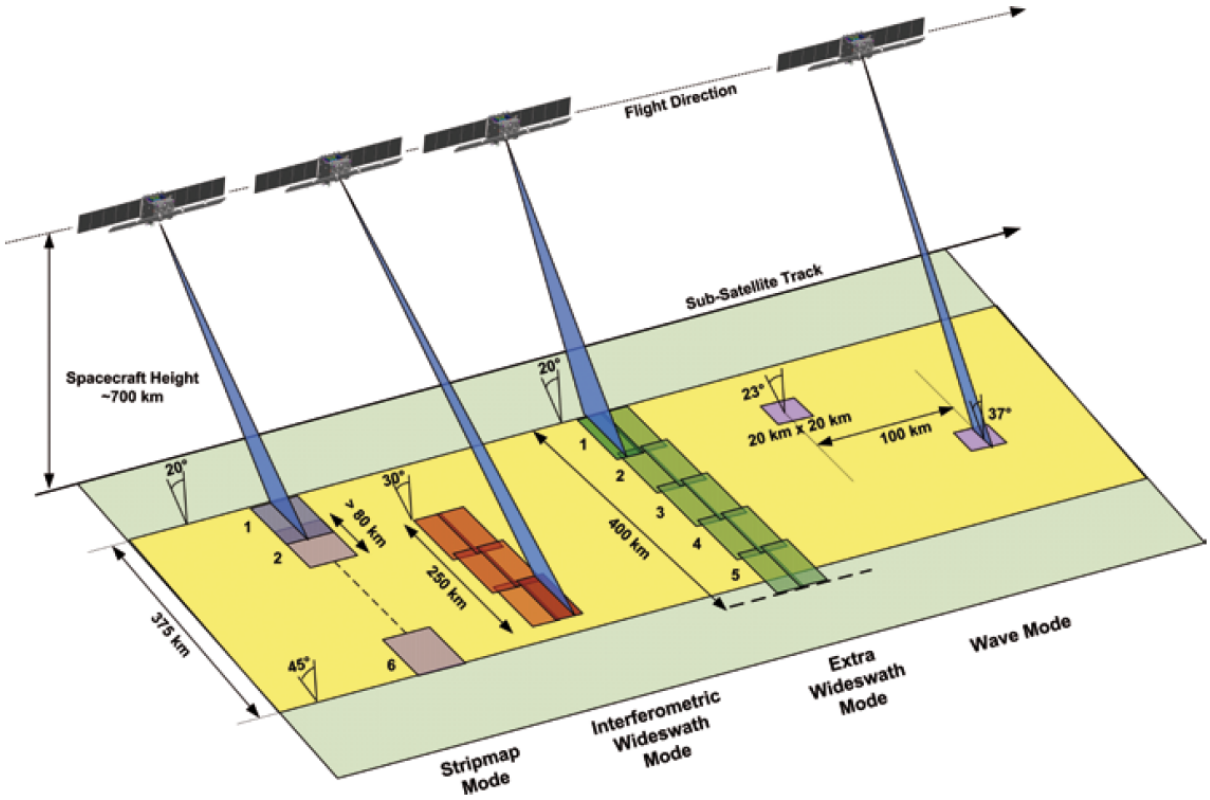

Abbildung 3: Aufnahmemodi der Sentinel-1 Mission [6]

Tabelle 2: Eigenschaften der Aufnahmemodi der Sentinel-1 Mission [6]

Modus	IW	WV	SM	EW
Polarisation	Dual	Single	Dual	Dual
Azimutauflösung (m)	20	5	5	40
Rage-Auflösung (m)	5	5	5	20
Streifenbreite (km)	250	20x20	80	410

Beide Satelliten befinden sich auf einem polnahen, sonnensynchronen Orbit. Ein Zyklus dauert 12 Tage, in denen die Erde 175 umrundet wird. Da er sich um ein Satellitenpaar handelt welches als Tandem die Erde umrundet wird ein Punkte alle sechs Tage von einem der Satelliten überflogen. Das System kann eine zuverlässige globale und systematische Abdeckung liefern. Dabei können im IW Modus alle relevanten Land-, Wasser- und Eismassen alle zwölf Tage vollständig von einem Satelliten erfasst werden. In Krisensituationen können nach Bedarf innerhalb von zweieinhalb und fünf Tagen Daten erfasst werden [6].
Nach dem Erfassen der Daten und Übersenden an eine Bodenstation werden diverse Vorverarbeitungsschritte vorgenommen in die sowohl interne also auch externe Parameter einfließen. Daraus ergeben sich diverse Produkte welche sich durch Aufnahmemodus (IW, SM, EW und

WV), Produkt-Typ sowie durch ihre Auflösung (Full-, High-, und Medium-Resolution) unterscheiden. Single Look Complex (SLC) Produkte sind im wesentlichen kalibrierte Rohdaten in denen Amplitude und Phase nicht zur Reflektivität kombiniert wurden und die geometrische Auflösung sich in Azimut- und Blickrichtung unterscheidet. Ground Rage Detected (GRD) Produkte bilden hingegen die Reflektivität ab und haben eine annähern quadratische geometrische Auflösung. Die Reflektivität wird in der logarithmischen Maßeinheit Dezibel (dB) angegeben. Die Korrektur der Schrägdistanz in Blickrichtung erfolgt durch Projektion auf einen Ellipsoiden. [8]. Aus den Level-1 Produkten, SLC und GRD, können die Level-2 Produkte OSW, OWI und RVL abgeleitet werden.

2.2.4 Datenzugang

Die Daten des Copernicus-Programmes sollen einer möglichst breiten Nutzergruppe möglichst einfach zugänglich gemacht werden. Sie sollen frei zugänglich und kostenlos angeboten werden [7]. Daten der Sentinel-1, 2, 3 und 5 können über das von der ESA betriebene Copernicus Open Access Hub bezogen werden. Datensätze können sowohl auf der Webseite als auch mithilfe einer API gesucht und heruntergeladen werden. Der Zugang zu Daten der Sentinel-3, 6 und 4 sowie weiterer Satelliten können über das dem Copernicus Open Access Hub ähnlichen EUMETCast bezogen werden. In Ergänzung zu diesen Quellen werden Daten von fünf privaten, in Kooperation mit dem Copernicus-Programm stehenden Unternehmen in unterschiedlichen Formen bereitgestellt. Diese als Data and Information Access Services (DIAS) bezeichneten Zugänge stellen unverarbeitete und abgeleitete Daten sowie Werkzeuge zur Analyse zur Verfügung [13]. Da die DIAS kommerziell betrieben werden müssen einige Dienste und Werkzeuge bezahlt werden während Nutzer sich lediglich am Copernicus Open Access Hub oder EUMETCast registrieren müssen. Zu erwähnen ist das die DIAS Zugriff auf die gesamten Daten gestatten. Aus dem Copernicus Open Access Hub lassen sich nur Teile der Daten synchron beziehen. In der Regel müssen Daten welche älter als einen Monat sind aus dem Archiv wiederhergestellt werden. Dieser Vorgang kann einige Zeit in Anspruch nehmen.

2.3 Überschwemmungsmonitoring

Um Wasserflächen und damit auch überflutete Areale auf Radarbildern zu erkennen können die Reflektionseigenschaften von Wasserflächen genutzt werden. Das Wasser eine sehr niedrige Rauigkeit besitzt kommt beim Aufprall eines Radarsignals zu einer spiegelnden Reflektion und nur ein sehr geringer Teil des Signals wird zum Empfänger zurückgeworfen. In den resultierenden Bildern äußert sich dieser Umstand in niedrigen Refelektivitätswerten. Um die Areale mit niedrigen Reflektionswerten zu detektieren können Verfahren genutzt werden, welche aus den Histogrammen der Bilder einen Schwellwert ermitteln. Um die Ergebnisse einer solchen Schwellwertbestimmung zu verbessern, sollten die Radardaten, zum Beispiel Sentinel-1 IW GRD, zusätzlich Kalibriert werden. So können die genaue Kenntnis über die tatsächliche

Flugbahn des Satelliten dazu betragen die geografische Genauigkeit zu verbessern. Diese kann zusätzlich durch Verfahren wie die Diffentialentzerrung gesteigert werden die die durch das Relief entstandenen Lagefehler ausgleicht [2]. Die radiometrische Genauigkeit kann gesteigert werden indem zum Beispiel thermisches Rauschen aus den Daten entfernt wird und die Reflektivitätswerte zum sogenannten σ_0-Wert umgerechnet werden. Dieser repräsentiert den Querschnitt der Reflektivität für eine normierte Fläche am Boden [11]. Dieses Maß erlaubt zudem das Vergleichen unterschiedlicher Radaraufnahmen. Auch sollte ein Speckle-Filter zum Einsatz kommen um. Dieser reduziert körnige Bildstrukturen welche auf homogenen Flächen in Radarbildern auftreten und die rechnerische Bildauswertung erschweren können. [2] [10]. Auf Basis des Schwellwertes kann ein Binärisierung des Bilder durchgeführt werden. Die entstehenden Werte würden überflutete beziehungsweise trocken liegende Areale repräsentieren [10]. Eines dieses Schwellwertverfahren wurde von Nobuyuki Otsu entwickelt und ist nach ihm benannt. Bei diesem Verfahren werden alle Werte eines Histogramms durchlaufen. Jeder dieser Werte teilt das Histogramm in zwei Gruppen und bildet so einen Schwellwert. Jener Wert welcher die gewichtete Varianz zwischen der Klassen maximiert wird als optimaler Grenzwert angesehen [9]. Gegeben sei ein Bild C mit N Pixeln in L Grauwertstufen. Die Anzahl der Pixel einer Grauwertstufe i sein dann gegeben durch n_i und es gilt:

$$N = \sum_{i=1}^{L} n_i \tag{1}$$

Ein betrachteter Grenzwert t teil das Bild in die Gruppen C_0 und C_1 wobei C_0 alle Pixel der Graustufen 1 bis t und C_1 alle Pixel der Graustufen $t+1$ bis L enthält. Die Gewichte für die Gruppen C_0 und C_1 sind nun gegeben durch:

$$w_0(t) = w(t) = \sum_{i=1}^{t} p_i \text{ und } w_1(t) = \sum_{i=t+1}^{L} p_i \tag{2}$$

mit p_i:

$$p_i = \frac{n_i}{N} \tag{3}$$

sowie μ_0, μ_1 und μ_T:

$$\mu_0(t) = \sum_{i=1}^{t} i p_i / w_0 \text{ und } \mu_1(t) = \sum_{i=t+1}^{L} i p_i / w_1 \text{ und } \mu_T = \sum_{i=1}^{L} i p_i \tag{4}$$

Die Klassenvarianzen $\sigma_0^2(t)$ und $\sigma_1^2(t)$ sind gegeben durch:

$$\sigma_0^2(t) = \sum_{i=1}^{t} (i-\mu_0)^2 p_i / w_0 \text{ und } \sigma_1^2(t) = \sum_{i=t+1}^{L} (i-\mu_1)^2 p_i / w_1 \tag{5}$$

Zu Maximieren ist nun die Inter-Klassenvarianz K:

$$K = \frac{\sigma_t^2}{\sigma_W^2} \tag{6}$$

mit σ_W^2 und σ_T^2:

$$\sigma_W^2 = w_0\sigma_0^2 + w_1\sigma_1^2 \text{ und } \sigma_T^2 = \sum_{i=1}^{L}(i-\mu_T)^2 p_i \tag{7}$$

Die Binärisierung kann direkt auf Basis der Radaraufnahme der Überflutung, oder auf abgeleiteten Daten erfolgen. So können zum Beispiel das Radaraufnahme der Überflutung σ_0^f mit einer überflutungsfreien Referenzaufnahme (σ_0^r) kombiniert zum Normalized Difference Sigma-Naught Index (NDSI) [12]. Dabei werden die Reflektivitätswerte von zwei unterschiedlichen Zeitpunkten zum NDSI verrechnet welcher als Maß für die Stärke der Veränderung interpretiert werden kann.

$$NDSI = \frac{\sigma_0^f - \sigma_0^r}{\sigma_0^f + \sigma_0^r} \tag{8}$$

Dieses Maß bewegt sich zwischen -1 und 1 wobei Werte um 0 für identische Reflektionswerte an beiden Zeitpunkten und daher für geringe Veränderung stehen. Aufgrund der Reflektionseigenschaften von Wasserflächen deuten Werte nahe -1 auf überflutete Areale hin [12]. Die der vielen und teilweise zeitintensiven Prozessierungsschritte können, je nach Größe des zu untersuchenden Areals, viel Zeit und Rechenleistung in Anspruch nehmen.

2.4 Web Application Programming Interfaces

Schnittstellen sind gemeinsame Grenzen zwischen funktionalen Einheiten über die mittels vorgegebener Kommunikationswege Informationen ausgetauscht werden können. Ein Application Programming Interface erlaubt also den Austausch von Informationen zwischen zwei unterschiedlichen Programmen. Genauer ausgedrückt erlaubt eine API einem Programm Funktionen eines anderen Programms zu nutzen [17]. Zu bemerken ist hierbei das keines der beiden Programme die programmatische Details des jeweils anderen kennt oder kennen muss [18]. Die Nutzung von APIs erlaubt die Modularisierung von Software in voneinander unabhängige, und untereinander austauschbare Module.
Bei diesen kann es sich zum Beispiel um Web-Anwendungen handeln welche ihre Informationen über standardisierte Protokolle, zum Beispiel HTTP über das Internet austauschen. Moderne Web-Anwendungen werden häufig nach dem REST Paradigma entwickelt. Dieses beschreibt einen Softwarearchitekturstil für verteilte Hypermedia Systeme wie das Internet. Durch die Umsetzung dieses Paradigmas sollen Webanwendungen Eigenschaften wie Skalierbarkeit, Ausfallsicherheit, Transparenz und Zuverlässigkeit. Außerdem sollen die Belange unterschiedlichen Anwendungen getrennt werden [18].

Zudem definiert das REST Paradigma fünf Einschränkungen bezüglich der Konfiguration von Anwendungen und deren Kommunikation. So werden Server klar von Clients abgegrenzt um die Skalierbarkeit der Server sicherzustellen. Stabile Schnittstellen zwischen Server und Client stellen zudem sicher das beide unabhängig voneinander entwickelt werden können. Um die Skalierbarkeit weiter zu steigern sollen Anwendungen stateless miteinander kommunizieren. Die bedeutet das alle Informationen die ein System zur Beantwortung eines Requests benötigt in selbigem enthalten sein müssen. Um die performante Beantwortung von Requests zu steigern soll es möglich sein Responses zu speichern und wiederzuverwenden. Zu guter Letzt sollen Anwendungen ressourcenbasiert arbeiten und diese mit wenigen, klar definierten Methoden abrufbar machen. Der Austausch von Ressourcen soll mit unterschiedlichsten Repräsentationen möglich sein. So kann eine Ressource zum Beispiel als .xml- oder .json-Datei verfügbar sein.

2.4.1 Rich Data Interfaces

Der Begriff Rich Data Interface kann unter zwei Gesichtspunkten betrachtet werden. Einerseits kann darunter eine API verstanden werden welche reich an Interaktionsmöglichkeiten ist, dem Nutzer also umfangreiche Funktionen zur Interaktion mit den durch die von der API angebotenen Ressourcen ermöglicht. Andererseits könnten die Ressourcen selber reich an Informationen sein. Reichhaltige Ressourcen können zum Beispiel vorprozsessierte Daten sein welche von Nutzern direkt für weitere Analysen verwendet werden können ohne das diese selber die möglicherweise Aufwendige Vorprozsessierung durchführen müssen.
Unter reichhaltigen Ressourcen können jedoch auch aus Rohdaten abgeleitete Produkte verstanden werden welche bestimmte, bereits in den Rohdaten vorhandene Aspekte hervorheben oder bereits Daten ableiten welche einen bestimmten Sachverhalt klar abbilden.

2.5 OGC und OGC Standards

Das Open Geospatial Consortium (OGC) widmet sich der Aufgabe die Entwicklung von internationalen Standards und unterstützender Dienste welche die Interoperabilität im Bereich der Geoinformatik verbessern voranzutreiben. Das OGC soll dabei offene Systeme und Techniken verbreiten welche es erlauben Dienste und Prozesse mit Raumbezug in Kreisen der Informatik verbreiten und die Nutzung von interoperabler und kommerzieller Software fördern [16]. Dabei wird versucht möglichst viel Akteure aus Wissenschaft, Wirtschaft und Verwaltung zu beteiligen um Standards zu schaffen welche auf möglichst breitem Konsens basieren. Zudem werden Mechanismen zur Zertifizierung von standardkonformen Software-Lösungen angeboten [16].
Das OGC formuliert Standards für unterschiedliche Themenbereiche. In Standards aus dem Bereich Data Models und Encodings werden Daten- und Datenaustauschformate definiert. Standards welche Webdienste und Schnittstellen zum Austausch von Geodaten beschreiben werden dem Bereich Services und APIs zugeordnet. Der OGC API - Processes - Part 1: Core Standard gehört in diesen Bereich. Die Bereiche Discovery und Containers enthalten Standards für die

Speicherung von Geodaten sowie die Auffindbarkeit und Durchsuchbarkeit dieser. Ein weiterer Bereich widmet sich Standards zum Thema Sensornetzwerke.

2.6 OGC API - Processes - Part 1: Core

Der OGC API - Processes - Part 1: Core Standard soll das Bereitstellen von aufwendigen Prozessierungsaufgaben und ausführbaren Prozessen welche über eine Web API von anderen Programmen aufgerufen und gestartet werden können unterstützen [14]. Der Standard ist dabei von Konzepten des OGC Web Processing Service 2.0 Interface Standards beeinflusst und bedient sich des REST Paradigmas sowie der Java Script Object Notation (JSON).
Der Aufbau des OGC API - Processes - Part 1: Core Standards orientiert sich am OGC Spezifikationsmodell. Dieses beschreibt die modularen Komponenten eines Standards und wie diese miteinander in Verbindung stehen [15]. Das Spezifikationsmodell definiert einen Standard als Teillösung eines Entwicklungsproblems. Diese Teillösung limitiert die Anzahl an möglichen Implementierungen. Ziel ist die Harmonisierung der Implementierungen und so die Interoperabilität zu steigern. Der OGC API - Processes - Part 1: Core Standard formuliert Requirements, Recommendations und fasst diese zu Requirements-Classes zusammen welche wiederum ein Standardisierungsziel beschreiben. Requirements beschrieben Eigenschaften oder Vorgehensweisen die die Implementierung umsetzen muss um standardkonform zu sein. Recommendations sind hingegen nicht verpflichtend beschreiben aber aus Sicht der Autoren empfehlenswerte Eigenschaften oder Vorgehensweisen [15]. Jedes Requirement kann mit einem ebenfalls im Standard definierten Conformance-Test-Case überprüft werden. Diese Tests können zu Conformance-Test-Modules zusammengefasst welche alle Test zum prüfen einer Requirements-Class umfassen. Die Gesamtheit dieser Conformance-Test-Modules wird auch als Conformance-Test-Class bezeichnet. Alle vom Standard Conformance-Test-Classes werden im Conformance-Suit zusammengefasst [15]. Erfüllt eine Implementierung alle im Conformance-Suit definierten Tests kann sie mit einem Certificate of Conformance für die implementierten Requirements-Classes versehen werden. Requirements, Recommendation und Conformance-Suit bilden gemeinsam eine Spezifikation welche nach der Anerkennung durch ein legitimes Gremium wie das OGC als Standard angesehen werden. Der OGC API - Processes - Part 1: Core Standard definiert sieben Requirements-Classes. Die Requirements-Class Core beschreibt dabei die Kernfunktionalitäten welche von standardkonformen Implementierungen umgesetzt werden. Da dem Nutzer mit diesen Kernfunktionalitäten Ressourcen zugänglich gemacht werden sollen werden in den Requirements-Classes JSON und HTML Repräsentationen dieser Ressourcen in JSON und HTML definiert [14]. Die Requirements-Class Core macht keine expliziten Vorgaben für die Beschreibung einer standardkonformen API. Solche Vorgaben finden sich in der nicht verpflichtend umzusetzenden Requirements-Class OpenAPI Specification 3.0. Diese definiert wie implementierte APIs mithilfe der OpenAPI 3.0 Spezifikation beschrieben und dokumentiert werden können [14]. Ebenso werden in der Core Requirements-Class keine expliziten Vorgaben

zur Beschreibung der angebotenen Prozesse gemacht. Da der Standard primär, aber nicht ausschließlich, zum Breitstellen von Diensten aus dem Bereich der Geoinformatik genutzt werden soll wird in der Requirements-Class OGC Process Description die Nutzung des OGC Processes Description Formats zum Beschreiben von angebotenen Prozessen empfohlen [14]. Zusätzliche Funktionen werden in den Requirements-Classes Job-List, Callback und Dismiss beschreiben. Sie beschreiben zusätzliche Ressourcen und Interaktionsmöglichkeiten mit den auszuführenden Instanzen eines Prozesses welche als Jobs bezeichnet werden [14]. Die Requirements-Classes definieren hier Endpoints sowie deren Funktionen, Responses und mögliche Fehlersituationen. Jede implementierte Requirements-Class kann mit einem entsprechenden Test, welcher im Abstract Test Suit beschreiben ist, auf ihre korrekte Implementierung hin überprüft werden.

2.7 Evaluationskriterien

Die Evaluation einer Implementierung einer API kann unter verschiedenen Gesichtspunkten erfolgen. Zum einen können technische Aspekte Effizienz, Skalierbarkeit, Stabilität und Wartbarkeit untersucht werden. Zu dieser technischen Bewertung einer API kann auch das Überprüfen der Standardkonformität gezählt werden. Diese kann durch einen ebenfalls zu implementierenden Unit-Test teilweise überprüft werden da mache Testfälle des Test-Suits des Standards automatisch überprüft werden können [14]. Der Fokus der in dieser Arbeit durchgeführt Evaluation soll jedoch die Benutzbarkeit und Benutzerfreundlichkeit der Implementierung sein. Dafür können die von Jakob Nielsen 1993 aufgestellten Heuristiken verwendet werden. Dabei handelt es sich um zehn Heuristiken unter denen eine Schnittstelle betrachtet werden kann [19]. Die Heuristiken decken unter anderem die Themenfelder Verständlichkeit, Fehlerbehandlung und Fehlervermeidung, Dokumentation und Konsistenz ab [19].

3 Implementierung

3.1 Softwarestack

Die prototypische Entwicklung eines leichtgewichtigen Rich Data Interface für Copernicus-Daten erfolgte im Rahmen dieser Arbeit mit der Programmiersprache Python. Diese ist nicht nur aufgrund ihrer Einfachheit vorteilhaft sondern erlaubt auch den Zugriff auf eine große Zahl von Packages für die unterschiedlichsten Anwendungsfälle. Weite Teile des Rich Data Interface wurden mithilfe des Flask-Frameworks umgesetzt. Dieses erlaubt das schnelle Entwickeln von kleinen und auf wenige Aufgaben fokussierten APIs.

3.2 Programmstruktur

Die Anwendung ist in vier Python-Scripte aufgeteilt. Im api.py Script ist die Webanwendung definiert. Neben Grundeinstellungen enthält diese Datei die Endpoints der API. Die geordnete Abarbeitung der angelegten Jobs werden vom Script processing.py gesteuert. Die eigentlichen Prozesse sowie Hilfsfunktionen befinden sich im utils.py Script. Diese Scripte verwalten Dateien in einem einfach Verzeichnissystem. Templates für statische Ressourcen befinden sich im Verzeichnis *templates/*. HTML-Dateien befinden sich im Verzeichnis *templates/html/* und JSON-Dateien im Verzeichnis *templates/json/*. Das Unterverzeichnis *templates/json/processes/* enthält die Beschreibungen der von der API angebotenen Prozesse. Die Anwendung erlaubt das persistente hinterlegen von Sentinel-1 Datensätzen um zeitaufwendiges Herunterladen zu vermeiden. Diese Datensätze können im Verzeichnis *data/* abgelegt werden. Jeder Sentinel-1 Datensatz enthält eine .kml-Datei welche Metadaten zum Datensatz enthält. Diese werden im Unterverzeichnis *data/coverage/* abgelegt. Jeder angelegt Job, also jede auszuführende Instanz einer Prozesses erhält ein einzigartiges Verzeichnis innerhalb des Verzeichnisses *jobs/*. In diesem Verzeichnis befinden sich eine Status- und Job-Datei sowie ein Footprint. Neben diesen Dateien enthält jedes Job-Verzeichnis ein Unterverzeichnis *results/* in dem die Ergebnisse des jeweiligen Jobs abgelegt werden.

3.3 Requirements Classes für Encodings

In der Requirements Class JSON wird definiert welche Ressourcen im Media-Type *application/json* angefragt werden können. Dazu gehören alle Responses der Endpunkte API Landing Page, API Definition, Conformance Deklaration, Prozess Liste, Prozess Beschreibung, Prozess Ausführung und Job Status welche mit dem HTTP-Statuscode 200 versandt werden. Da die prototypische Implementierung auch die Endpunkte Job Liste und Coverage bereitstellt können die korrespondierenden Ressourcen auch im Media-Type *application/json* angefragt werden. Werden Ressourcen im Media-Type *application/json* angefragt müssen also zunächst die entsprechenden .json-Dateien geladen, je nach Endpunkt variiert oder kombiniert und vor dem

versenden mit der Funktion *jsonify()* vorbereitet werden.
In der Requirements-Class HTML werden analog zu Requirements Class JSON jene Ressourcen definiert welche im Media-Type *text/html* angefragt werden können. Jedoch entfällt in dieser Requirements-Class die Einschränkung auf bestimmte Endpunkte und alle Responses welche mit dem HTTP-Statuscode 200 versandt werden müssen den Media-Type *text/html* unterstützen. Wird dieser angefragt wird die entsprechende Ressource mit der Funktion *render_template()* gerendert und als Response versandt. Dabei können zusätzliche Elemente übergeben werden welche von der in Flask enthaltenen Jinja template engine dynamisch dargestellt werden können. Stellen Endpoints ihre Ressourcen sowohl den Media-Type *application/json* als auch *text/html* zur Verfügung so können Nutzer diesen über den optional Parameter *f* oder *content_type* spezifizieren. Wird kein Media-Type über diese Parameter spezifiziert so wird standardmäßig der Media-Type *text/html* verwendet.
Jede Ressource die in beiden Media-Typen bereitgestellt wird enthält eine Verknüpfungen zu sich selbst mit der Relation *self* und eine Verknüpfung zur Ressource im jeweils anderen Media-Type mit der Relation *alternate*.

3.4 Requirements Class Core

3.4.1 HTTP 1.1

Die Umsetzung der Requirements-Class HTTP 1.1 (RFC 2616) verlangt das die API exklusiv das HTTP 1.1 unterstützt. Falls die API ebenfalls HTTPS unterstützt muss ebenfalls HTTP over TLS (RFC 2818) eingehalten werden. Das Flask-Framework nutzt standardmäßig das HTTP 1.0. Teil des Flask-Frameworks ist die WSGI Bibliothek Werkzeug welche das Implementieren von Webanwendungen erlaubt. Um die verwendete HTTP-Version von 1.0 auf 1.1 umzustellen müssen Variablen in Werkzeug angepasst werden. Nach dem Import der Module WSGIRequestHandler und BaseWSGIServer kann in beiden die Version des HTTP Protokolls angepasst werden (siehe Anhang A.1).
In dieser Requirements-Class werden zudem alle HTTP-Statuscodes gelistet die Nutzer von einer standardkonformen Implementierung mindestens erwarten können.

Tabelle 3: Vorgesehene HTTP-Statuscodes [14]

HTTP-Statuscode	Bedeutung
200	OK
201	Created
204	No Content
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
410	Gone
429	Too Many Requests
500	Internal Server Error
501	Not Implemented

Alle erfolgreichen Anfragen welche eine Resource liefern mit dem HTTP-Statuscode 200 beantwortet. Die Verwendung nicht zulässiger HTTP-Methoden resultieren in Antworten mit dem Status-Code 405 während Anfragen für nicht unterstütze Media-Types mit dem Status-Code 406 beantwortet werden. Kommt es zu Fehlern bei der Ausführung des Programmcodes antwortet die Anwendung mit dem HTTP-Statuscode 500. Werden durch eine Anfrage Ressourcen neu erzeugt oder nicht gefunden antwortet die Anwendung mit den HTTP-Statuscodes 201 beziehungsweise 404. Der Standard erlaubt die Nutzung weiterer HTTP-Statuscodes [14].

3.4.2 Limit Paramter

Der *limit*-Parameter wird von den Endpoints Process List und Job List unterstützt. Mit ihm kann gesteuert werden wie viele Elemente im Response gelistet werden. Der *limit*-Parameter ist optional. Ist er nicht Teil eines Requests werden standardmäßig 10 Elemente zurückgegeben. Es können maximal 1000 Elemente angefragt werden. Ergibt die Überprüfung des *limit*-Parameters das Werte außerhalb des gültigen Wertebereichs von 0 bis 1000 angefragt werden wird der Parameter auf 10 zurückgesetzt. Ein Response kann weniger, aber nie mehr Elemente als durch den *limit*-Parameter spezifiziert werden enthalten B.3.

3.4.3 API Landig Page

Der erste im Standard definierte Endpoint kann über den URL*http://HOST:PORT/?f=<MEDIA-TYPE>* aufgerufen werden und liefert als Resource die Landing Page der API. Der Endpoint kann nur mit der HTTP-Get Methode verwendet werden.

Diese kann als Eintrittspunkt zu allen anderen Funktionalitäten angesehen werden. Sie enthält Verknüpfungen zu den Endpoints API Definition, Conformance Declaration, Process List, Process Description, Job List und Coverage. Die Resource kann in den Media-Types *text/html* und *application/json* abgerufen werden.
Wird ein Request akzeptiert so wird, je nach gewähltem Media-Type, ein Response aus einer statischen .html- oder .json-Datei generiert und mit zusätzlichen HTTP-Headern versandt. Der *link*-Header liefert einen URL zur angefragten Resource während der *resource*-Header die angefragte Ressource identifiziert (siehe Anhang A.2).

3.4.4 API Definition

Unter dem URL *http://HOST:PORT/api?f=<MEDIA-TYPE>* kann der API Definition Endpoint erreicht werden. Auch dieser Endpoint erlaubt nur die Nutzung der HTTP-Get Methode.

Dieser Endpoint liefert eine detaillierte Beschreibung der API und ihrer Funktionen. Da eine andere Requirements-Class die Dokumentation im OpenAPI 3.0 Format fordert sind die von diesem Endpunkt bereitgestellten Ressourcen aus einer solchen abgeleitet. In dieser Form der Dokumentation werden neben einer allgemeinen Beschreibung sämtliche Endpoints mit ihren zu erwartenden HTTP-Statuscodes und Responses beschrieben. Zusätzlich erfolgt einen Auflistung aller Schemata nach welchen von der API angebotene Ressourcen strukturiert sind.

Bei einem gültigen Request werden die angefragten Ressourcen werden auch hier aus statischen .html- beziehungsweise .json-Dateien generiert und werden zusammen mit einem *link*- und *resource*-Header versandt (siehe Anhang A.3).

3.4.5 Conformance Endpoint

Der Conformance Endpoint kann unter dem URL *http://HOST:PORT/conformance?f=<MEDIA-TYPE>* angefragt werden. Für Requests an diesen Endpoint ist allein die HTTP-Get Methode zulässig.

Stellt ein Nutzer einen Request an den Conformance Endpoint so erhält er Informationen zur Konformität der API zum OGC API - Processes - Part 1: Core Standard. In der Ressource werden Verknüpfungen zu allen Requirements-Classes gelistet welche von der API implementiert werden.

Der Response wird nach einem gültigen Request aus den statischen .html beziehungsweise .json Dateien erzeugt und samt *link*- und *resource*-Header versandt (siehe Anhang A.4).

3.4.6 Processes Endpoint

Requests an den Process List endpoint müssen an den URL *http://HOST:PORT/processes?f=<MEDIA-TYPE>&limit=<INTEGER>* und der HTTP-Get Methode gesendet werden.

Als Resource liefert dieser Endpoint eine detaillierte Liste der angebotenen Prozesse als HTML- oder JSON-Dokument. In dieser Liste werden die Bezeichnung, die Steueroptionen sowie die Ein- und Ausgaben jedes angebotenen Prozesses gelistet.

Der Process List Endpoint unterstützt den *limit*-Parameter welcher die Anzahl der wiedergegebene Prozesse steuert.

Wir ein gültiger Request an diesen Endpoint gestellt werden zunächst alle Process Descriptions aus dem Verzeichnis */templates/json/processes* geladen und in einem Array gespeichert. Wird der Media-Type *aplication/json* angefragt wird das Array mit der vom *limit*-Parameter festgelegten Länge und den Verknüpfungen *self* und *alternate* an die Funktion *jsonify()* übergeben und als Response versandt. Eine ähnliche Verfahrensweise kommt zum Einsatz wenn der Media-Type *text/html* angefragt wurde. In diesem Fall wird das durch den *limit*-Parameter in seiner Länge eingeschränkte Array samt dem entsprechenden HTML-Template an den Renderer übergeben und anschließend als Response versandt (siehe Anhang A.5).

3.4.7 Process Endpoint

Der Process Description Endpoint kann über den URL *http://HOST:PORT/processes/<processID>?f=<MEDI*
TYPE> ausschließlich mit der HTTP-Get Methode angefragt werden.

Dieser Endpoint liefert als Response detaillierte Informationen zu dem im Request über seine eindeutige Bezeichnung spezifizierten Prozess.

3.4.8 Prozess Ausführung

3.4.9 Job Status

3.4.10 Job Resultate

3.5 Requirements Class OGC Process Description

3.6 Requirements Class Job List

3.7 Requirements Class Dismiss

3.8 Requirements Class OpenAPI 3.0

Der OGC API - Processes - Part 1: Core Standard macht über die eigentlichen Funktionen der API auch Vorgaben zur Art der Dokumentation der API. Hierfür soll der OpenAPI 3.0 Standard genutzt werden.

3.9 Prozesse

3.9.1 Echo

Da eine standardkonforme API mindestens einen testbaren Prozess anbieten muss ist der Echo-Prozess ebenfalls Teil der prototypischen Implementierung. Dieser nimmt einen beliebigen Wert entgegen. Nach einer kurzen, simulierten Bearbeitungszeit kann dieser wieder als Resultat abgefragt werden.
Nach dem Start eines Echo Prozesses wird zunächst überprüft ob der Job nicht den Status *dismissed* aufweist. Wäre dies der Fall wird die Ausführung abgebrochen. Andernfalls wird der *started*-Eintrag in der status.json mit dem aktuellen Zeitstempel versehen und der zurückzugebende Wert aus den Eingaben des Jobs gelesen. Schlägt dies fehl wird der *status*-Eintrag in der status.json auf *failed* gesetzt und der Ausführung abgebrochen. Anschließend wartet das Programm fünf Sekunden. Nach einer erneuten Prüfung des Job-Status wird das Ergebnis als .json in das *results/*-Verzeichnis des Jobs geschrieben. Diese results.json enthält den Eingabewert und die Nachricht das es sich um ein Echo handelt. Als letzter Schritt wird der Job-Status, der Fortschritt, der Infotext sowie der Beendigungszeitpunkt in der Status-Datei des Jobs aktualisiert A.8.

3.9.2 Überflutungsmonitoring

3.10 Zusätzliche Funktionalitäten

3.10.1 Coverage

4 Evaluation

4.1 Nielsen's Nutzbarkeitsheuristiken

4.2 Unit Testsuit

4.3 OGC Compliance Test

5 Diskussion

6 Ausblick

7 Fazit

Literatur

[1] A. Moreira, M. Younis, P. Prats-Iraola, G. Krieger, I. Hajnsek und K. P. Papathanassiou (2013, April 17). A Tutorial on Synthetic Aperture Radar [Online]. Verfügbar unter: https://www.researchgate.net/publication/257008464_A_Tutorial_on_Synthetic_Aperture_Radar (Zugriff am: 6. Juni 2022).

[2] J. Albertz, Einführung in die Fernerkundung, 4. Auflage Darmstadt: Wissenschaftliche Buchgesellschaft, 2009

[3] Europäische Kommission (2018, Oktober 06). Copernicus: 20 years of History [Online]. Verfügbar unter: https://www.copernicus.eu/en/documentation/information-material/signature-esafrance-collaborative-ground-segment (Zugriff am: 13. Juni 2022).

[4] European Space Agency (2018). Sentinels - Space for Copernicus [Online]. Verfügbar unter: https://www.d-copernicus.de/daten/satelliten/daten-sentinels/ (Zugriff am: 13. Juni 2022).

[5] Europäische Kommission (2019). What is Copernicus [Online]. Verfügbar unter: https://www.copernicus.eu/en/documentation/information-material/brochuresbrochures (Zugriff am: 13. Juni 2022).

[6] ESA Communications (2012, März). Sentinel-1 ESA's Radar Observatory Mission for GMES Operational Services [Online]. Verfügbar unter: https://sentinel.esa.int/web/sentinel/missions/sentinel-1/overview (Zugriff am: 13. Juni 2022).

[7] Europäisches Parlament und Rat der Europäischen Union (2014, April 24). Regulation (EU) No 377/2014 Establishing the Copernicus Programme and repealing Regulation (EU) No 911/2010 [Online]. Verfügbar unter: https://www.kowi.de/Portaldata/2/Resources/horizon2020/coop/Copernicus-regulation.pdf (Zugriff am: 13. Juni 2022).

[8] M. Bourbigot, H. Johnson, R. Piantanida (2016, März 03). Sentinel-1 Product Definition [Online]. Verfügbar unter: https://sentinel.esa.int/web/sentinel/user-guides/sentinel-1-sar/document-library/-/asset_publisher/1dO7RF5fJMbd/content/sentinel-1-product-definition (Zugriff am: 13. Juni 2022).

[9] N. Otsu (1976, Januar). A Threshold Selection Method from Gray-Level Histograms [Online]. Verfügbar unter: https://ieeexplore.ieee.org/document/4310076/citations#citations (Zugriff am: 14. Juni 2022).

[10] A. McVittie (2019, Februar). Sentinel-1 Flood mapping tutorial [Online]. Verfügbar unter: https://step.esa.int/main/doc/tutorials/ (Zugriff am: 15. Juni 2022).

[11] N. Miranda und P.J. Meadows (2015, Mai 21). Radiometric Calibration of S-1 Level-1 Products Generated by the S-1 IPF [Online]. Verfügbar unter: https://sentinel.esa.int/web/sentinel/user-guides/document-library/-/asset_publisher/xlslt4309D5h/content/sentinel-1-radiometric-calibration-of-products-generated-by-the-s1-ipf (Zugriff am: 15. Juni 2022).

[12] N. I. Ulloa, S.-H. Chiang und S.-H. Yun (2020, April 27). Flood Proxy Mapping with Normalized Difference Sigma-Naught Index and Shannon's Entropy [Online]. Verfügbar unter: https://doi.org/10.3390/rs12091384 (Zugriff am: 21. Juni 2022).

[13] Europäische Kommission (2018, Juni). The DIAS: User-friendly Access to Copernicus Data and Information [Online]. Verfügbar unter: https://www.copernicus.eu/en/access-data/dias (Zugriff am: 24. Juni 2022)

[14] B. Pross und P. A. Vretanos. (2021, Dezember 20). OGC API – Processes – Part 1: Core [Online]. Verfügbar unter: https://docs.opengeospatial.org/is/18-062r2/18-062r2.html (Zugriff am: 24. Juni 2022).

[15] S. Cox, D. Danko, J. Greenwood, J.R. Herring, A. Matheus, R. Pearsall, C. Portele, B. Reff, P. Scarponcini, A. Whiteside (2009, Oktober 19). The Specification Model - A Standard for Modular specifications [Online]. Verfügbar unter: https://www.ogc.org/standards/modularspec (Zugriff am: 27. Juni 2022).

[16] Open Geospatial Consortium (2021, Dezember 16). Bylaws of Open Geospatial Consortium [Online]. Verfügbar unter: https://www.ogc.org/ogc/policies (Zugriff am: 27. Juni 2022).

[17] C. Holmes, D. WWesloh, C. Heazel, G. Gale, A. Christl, J. Lieberman, C. Reed, J. Herring, M. Desruisseaux, D. Blodgett, S. Simmons, B. de Lathower und G. Percivall (2017, Februar 23). OGC Open Geospatial APIs - White Paper [Online]. Verfügbar unter: https://docs.ogc.org/wp/16-019r4/16-019r4.html (Zugriff am: 03. Juli 2022).

[18] F. Houbie, s. Sankaran, J. Lieberman, P. Vretanos, J. Masó (2016, Januar 16). OGC Testbed 11 REST Interface Engineering Report [Online]. Verfügbar unter: https://www.ogc.org/docs/er (Zugriff am: 05. Juli 2022).

[19] J. Nielsen, Usability Engineering, Mountain View: Academic Press Inc., 1993

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit zum Thema Rich Data Interfaces for Copenicus Data selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Münster, den 5. Juli 2022

A Quellcodeverzeichnis

A.1 Konfiguration von Werkzeug auf HTTP 1.1

Quellcode 1: Konfiguration von Werkzeug auf HTTP 1.1

```
    from flask import Flask
    from werkzeug.serving import WSGIRequestHandler
    from werkzeug.serving import BaseWSGIServer
    WSGIRequestHandler.protocol_version = "HTTP/1.1"
    BaseWSGIServer.protocol_version = "HTTP/1.1"
```

A.2 Quellcode Landing Page Endpoint

Quellcode 2: Landing Page Endpoint

```
#landingpage endpoint
@app.route('/',  methods = ['GET'])
def getLandingPage():
    app.logger.info('/')
    try:
        if(request.content_type == "text/html" or
        request.args.get('f')=="text/html" or
        request.args.get('f') == None):
                response = render_template('html/landingPage.html')
                return response, 200, {
                "link": "localhost:5000/?f=text/html",
                "resource": "landingPage"
                }
        elif(request.content_type == "application/json" or
        request.args.get('f')=="application/json"):
             file =  open('templates/json/landingPage.json',)
            payload = json.load( file)
             file.close()
            response = jsonify(payload)
            return response, 200, {
                "link": "localhost:5000/?f=application/json",
                "resource": "landingPage"}
        else:
            return "HTTP status code 406: not acceptable", 406
        except:
            return "HTTP status code 500: internal server error", 500
```

A.3 Quellcode API Definition Endpoint

Quellcode 3: API Definition Endpoint

```
#api endpoint
@app.route('/api',  methods = ['GET'])
def getAPIDefinition():
    app.logger.info('/api')
    try:
        if(request.content_type == "text/html" or
        request.args.get('f')=="text/html" or
        request.args.get('f') == None):
            response = render_template('html/apiDefinition.html')
            return response, 200, {
                "link": "localhost:5000/apiDefinition?f=text/html",
                "resource": "apiDefinition"}
        elif(request.content_type == "application/json" or
        request.args.get('f')=="application/json"):
             file =  open('templates/json/apiDefinition.json',)
            payload = json.load( file)
             file.close() #close apiDefinition.json
            response = jsonify(payload)
            return response, 200, {
                "link": "localhost:5000/api?f=application/json",
                "resource": "apiDefinition"}
        else:
            return "HTTP status code 406: not acceptable", 406
    except:
        return "HTTP status code 500: internal server error", 500
```

A.4 Quellcode Conformance Endpoint

Quellcode 4: Conformance Endpoint

```
#conformance endpoint
@app.route('/conformance',  methods = ['GET'])
def getConformance():
    app.logger.info('/conformance')
    try:
        if(request.content_type == "text/html" or
        request.args.get('f')=="text/html" or
        request.args.get('f') == None):
            response = render_template('html/confClasses.html')
            return response, 200, {
                "link": "localhost:5000/conformance?f=text/html",
                "resource": "conformance"}
        elif(request.content_type == "application/json" or
        request.args.get('f')=="application/json"):
             file =  open('templates/json/confClasses.json',)
            payload = json.load( file)
             file.close()
            response = jsonify(payload)
            return response, 200, {
                "link": "localhost:5000/conformance?f=application/json",
                "resource": "conformance"}
        else:
            return "HTTP status code 406: not acceptable", 406
    except:
        return "HTTP status code 500: internal server error", 500
```

A.5 Quellcode Process List Endpoint

Quellcode 5: Process List Endpoint

```
#processes endpoint
@app.route('/processes', methods = ['GET'])
def getProcesses():
    app.logger.info('/processes')
    if(request.args.get('limit') == None or
         int(request.args.get('limit')) <= 0 or
         int(request.args.get('limit')) > 1000):
        limit = 10 #set limit to default value
    else:
        limit =  int(request.args.get('limit'))
    try:
        if(request.content_type == "text/html" or
            request.args.get('f')=="text/html" or
            request.args.get('f') == None):
                processList = [] #initialize list of processes
                processDescriptions = os.listdir("templates/json/processes")
                counter = 0
                for i in processDescriptions:
                     file =  open('templates/json/processes/' + i,)
                    process = json.load( file)
                     file.close()
                    processList.append(process)
                    counter += 1
                    if(counter == limit):
                        break
                response = render_template('html/processes.html',
                    processes=processList)
                return response, 200, {
                    "link": "localhost:5000/processes?f=text/html",
                    "resource": "processes"}
        elif(request.content_type == "application/json" or
                 request.args.get('f')=="application/json"):
            processList = [] #initialize list of processes
            processDescriptions = os.listdir("templates/json/processes")
            for i in processDescriptions:
                 file =  open('templates/json/processes/' + i,)
                process = json.load( file)
                 file.close()
                processList.append(process)
            processes = {"processes": processList[0:limit],
                        "links": [ #add links to self and alternate
                            {
                            "href": "localhost:5000/processes?f=applicattion/json",
                            "rel": "self",
                            "type": "application/json"
                            },
                            {
                            "href": "localhost:5000/processes?f=text/html",
                            "rel": "alternate",
                            "type": "text/html"
                            }
                        ]}
            response = jsonify(processes)
            return response, 200, {
                "link": "localhost:5000/processes?f=application/json",
```

```
56                  "resource": "processes"}
57          else:
58              return "HTTP status code 406: not acceptable", 406
59      except:
60          return "HTTP status code 500: internal server error", 500
```

A.6 Quellcode Process Description Endpoint

Quellcode 6: Process Description Endpoint

```
#process endpoint
@app.route('/processes/<processID>', methods = ['GET'])
def getProcess(processID):
    app.logger.info('/processes/' + processID)
    try:
        if(request.content_type == "text/html" or
            request.args.get('f')=="text/html" or
            request.args.get('f') == None):
            if(os.path.exists('templates/json/processes/'
            +  str(processID) + 'ProcessDescription.json')):
                 file =  open('templates/json/processes/'
                +  str(processID)
                + 'ProcessDescription.json',)
                process = json.load( file)
                 file.close()
                response = render_template("html/Process.html", process=process)
                return response, 200, {"link": "localhost:5000/processes/"
                +  str(processID)
                + "?f=text/html",
                "resource":  str(processID)}
            else:
                exception = render_template('html/exception.html',
                title="No such process exception",
                description="Requested process could not be found",
                 type="no-such-process")
                return exception, 404, {"resource": "no-such-process"}
        elif(request.content_type == "application/json" or
                request.args.get('f')=="application/json"):
            if(os.path.exists('templates/json/processes/'
            +  str(processID)
            + 'ProcessDescription.json')):
                 file =  open('templates/json/processes/'
                +  str(processID)
                + 'ProcessDescription.json',)
                payload = json.load( file)
                 file.close()
                response = jsonify(payload)
                return response, 200, {"link": "localhost:5000/processes/"
                +  str(processID)
                + "?f=application/json",
                "resource":  str(processID)}
            else:
                exception = {"title": "No such process exception",
                "description": "Requested process could not be found",
                "type": "no-such-process"}
                return exception, 404, {"resource": "no-such-process"}
        else:
            return "HTTP status code 406: not acceptable", 406
    except:
        return "HTTP status code 500: internal server error", 500
```

A.7 Quellcode Process Execution Endpoint

Quellcode 7: Process Execution

```
@app.route('/processes/<processID>/execution', methods = ['POST'])
def executeProcess(processID):
    app.logger.info('/processes/' + processID + '/execution')
    try:
        if(os.path.exists('templates/json/processes/'
        +  str(processID)
        + 'ProcessDescription.json')):
            data = json.loads(request.data.decode('utf8').replace("'", '"'))
            inputParameters = utils.parseInput(processID, data)
            if(inputParameters == False):
                return "HTTP status code 400: bad request", 400
            jobID =  str(uuid.uuid4())
            created =  str(datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
            os.mkdir("jobs/" + jobID)
            os.mkdir("jobs/" + jobID + "/results/")

            job_file = {"jobID":  str(jobID),
                        "processID":  str(processID),
                        "input": inputParameters[0],
                        "output": inputParameters[2],
                        "responseType": inputParameters[1],
                        "path": "jobs/" + jobID,
                        "results": "jobs/" + jobID + "/results/",
                        "downloads": "jobs/" + jobID + "/downloads/"}
            json.dumps(job_file, indent=4)
            with  open("jobs/" + jobID + "/job.json", 'w') as f:
                json.dump(job_file, f)
            f.close()

            status_file = {"jobID":  str(jobID),
                           "processID":  str(processID),
                           "status": "accepted",
                           "message": "Step 0/1",
                           "type": "process",
                           "progress": 0,
                           "created": created,
                           "started": "none",
                           "finished": "none",
                           "links": [
                               {
                               "href": "localhost:5000/jobs/"
                               + jobID + "?f=application/json",
                               "rel": "self",
                               "type": "application/json",
                               "title": "this document as JSON"},
                               {
                               "href": "localhost:5000/jobs/"
                               + jobID + "?f=text/html",
                               "rel": "alternate",
                               "type": "text/html",
                               "title": "this document as HTML"
                               }
                          ]}
            json.dumps(status_file, indent=4) #dump content
            with  open("jobs/" + jobID + "/status.json", 'w') as f: #create file
```

```
56                  json.dump(status_file, f) #write content
57              f.close() #close file
58
59              response = jsonify(status_file) #create response
60              return response, 201, {"location": "localhost:5000/jobs/"
61              + jobID + "?f=application/json",
62              "resource": "job"}
63              exception = {"title": "No such process exception",
64              "description": "Requested process could not be found",
65              "type": "no-such-process"}
66              return exception, 404
67      except:
68          return "HTTP status code 500: internal server error", 500
```

A.8 Quellcode Echo Process

Quellcode 8: Echo Process

```
def echoProcess(job):
    if(checkForDismissal(job.path + '/status.json') == True):
        return

    setStarted(job.path + '/status.json')

    try:
         input = job. input[0]
        time.sleep(5)
    except:
        updateStatus(job.path + '/status.json', "failed", "The job has failed", "-")
        return

    if(checkForDismissal(job.path + '/status.json') == True):
        return

    result ={"result":  input,
             "message": "This is an echo"}
    json.dumps(result, indent=4)
    with  open(job.results + "result.json", 'w') as f: #create file
        json.dump(result, f) #write content
        f.close() #close file
    updateStatus(job.path + '/status.json', "successful", "Step 1 of 1 completed", "100")
    setFinished(job.path + '/status.json')
```

B Schemata

B.1 landingPage.yaml

Schema 9: landingPage.yaml

```
type: object
required:
  - links
properties:
  title:
    type: string
    example: Example processing server
  description:
    type: string
    example: Example server
  links:
    type: array
    items:
```

```
14        $ref: "link.yaml"
```

B.2 processList.yaml

Schema 10: processList.yaml

```
1  type: object
2  required:
3    - processes
4    - links
5  properties:
6    processes:
7      type: array
8      items:
9        $ref: "processSummary.yaml"
10   links:
11     type: array
12     items:
13       $ref: "link.yaml"
```

B.3 limit.yaml

Schema 11: limit.yaml

```
1  name: limit
2  in: query
3  required: false
4  schema:
5    type: integer
6    minimum: 1
7    maximum: 10000
8    default: 10
9  style: form
10 explode: false
```

B.4 link.yaml

Schema 12: link.yaml

```yaml
type: object
required:
  - href
properties:
  href:
    type: string
  rel:
    type: string
    example: service
  type:
    type: string
    example: application/json
  hreflang:
    type: string
    example: en
  title:
    type: string
```

C Ressourcen

C.1 landingPage.html

Ressource 13: landingPage.html

```html
<!DOCTYPE html>
<html>
  <body>
  <h1>links:</h1>
    <p>
      href:<a href="localhost:5000/?f=text/html">
            localhost:5000/?f=text/html</a><br>
      rel: self<br>
      type: text/html<br>
      title: This document as HTML
    </p>
    <p>
      href:<a href="localhost:5000/?f=application/json">
            localhost:5000/?f=application/json</a><br>
      rel: alternate<br>
      type: application/json<br>
      title: This document as JSON
    </p>
    <p>
      href:<a href="localhost:5000/api?f=application/json">
            localhost:5000/apiDefinition?f=application/json</a><br>
      rel: service-desc<br>
      type: application/json<br>
```

```
      title: API definition for this endpoint as JSON
    </p>
    <p>
      href:<a href="localhost:5000/api?f=text/html">
            localhost:5000/apiDefinition?f=text/html</a><br>
      rel: service-desc<br>
      type: text/html<br>
      title: API definition for this endpoint as HTML
    </p>
    <p>
      href:<a href="localhost:5000/conformance?f=application/json">
            localhost:5000/conformance?f=application/json</a><br>
      rel: conformance<br>
      type: application/json<br>
      title: OGC API - Processes conformance classes implemented by this server as JSON
    </p>
    <p>
      href:<a href="localhost:5000/conformance?f=text/html">
            localhost:5000/conformance?f=text/html</a><br>
      rel: conformance<br>
      type: text/html<br>
      title: OGC API - Processes conformance classes implemented by this server as HTML
    </p>
    <p>
      href:<a href="localhost:5000/processes?f=application/json">
            localhost:5000/processes?f=application/json</a><br>
      rel: processes<br>
      type: application/json<br>
      title: Metadata about the processes as JSON
    </p>
    <p>
      href:<a href="localhost:5000/processes?f=text/html">
            localhost:5000/processes?f=text/html</a><br>
      rel: processes<br>
      type: text/html,<br>
      title: Metadata about the processes as HTML
    </p>
    <p>
      href:<a href="localhost:5000/jobs?f=application/json">
            localhost:5000/jobs?f=application/json</a><br>
      rel: jobs<br>
      type: application/json<br>
      title: The endpoint for job monitoring as JSON
    </p>
    <p>
      href:<a href="localhost:5000/jobs?f=text/html">
            localhost:5000/jobs?f=text/html</a><br>
      rel: jobs<br>
      type: text/html<br>
      title: The endpoint for job monitoring as HTML
    </p>
    <p>
      href:<a href="localhost:5000/coverage?f=application/json">
            localhost:5000/coverage?f=application/json</a><br>
      rel: coverage<br>
      type: application/json<br>
      title: The endpoint for coverage as JSON
    </p>
    <p>
```

```
      href:<a href="localhost:5000/coverage?f=text/html">
            localhost:5000/coverage?f=text/html</a><br>
      rel: coverage<br>
      type: text/html<br>
      title: The endpoint for coverage as HTML
    </p>
  </body>
</html>
```

C.2 landingPage.json

Ressource 14: landingPage.json

```
{
    "links": [
    {
        "href": "localhost:5000/?f=application/json",
        "rel": "self",
        "type": "application/json",
        "title": "This document"
    },{
        "href": "localhost:5000/?f=text/html",
        "rel": "alternate",
        "type": "text/html",
        "title": "This document as HTML"
    },
    {
        "href": "localhost:5000/api?f=application/json",
        "rel": "service-desc",
        "type": "application/json",
        "title": "API definition for this endpoint as JSON"
    },
    {
        "href": "localhost:5000/api?f=text/html",
        "rel": "service-desc",
        "type": "text/html",
        "title": "API definition for this endpoint as HTML"
    },
    {
        "href": "localhost:5000/conformance?f=application/json",
        "rel": "conformance",
        "type": "application/json",
        "title": "OGC API - Processes conformance classes implemented by this server as
            JSON"
    },
    {
        "href": "localhost:5000/conformance?f=text/html",
        "rel": "conformance",
        "type": "text/html",
        "title": "OGC API - Processes conformance classes implemented by this server as
            HTML"
    },
    {
        "href": "localhost:5000/processes?f=application/json",
        "rel": "processes",
        "type": "application/json",
        "title": "Metadata about the processes as JSON"
```

```
    },
        {
        "href": "localhost:5000/processes?f=text/html",
        "rel": "processes",
        "type": "text/html",
        "title": "Metadata about the processes as HTML"
    },
    {
        "href": "localhost:5000/jobs?f=application/json",
        "rel": "jobs",
        "type": "application/json",
        "title": "The endpoint for job monitoring as JSON"
    },
        {
        "href": "localhost:5000/jobs?f=text/html",
        "rel": "jobs",
        "type": "text/html",
        "title": "The endpoint for job monitoring as HTML"
    },
    {
        "href": "localhost:5000/coverage?f=application/json",
        "rel": "coverage",
        "type": "application/json",
        "title": "The endpoint for coverage as JSON"
    },
        {
        "href": "localhost:5000/coverage?f=text/html",
        "rel": "coverage",
        "type": "text/html",
        "title": "The endpoint for coverage as HTML"
    }
    ]
}
```