Integration und Austausch von Daten zu Bodenstationen innerhalb des DLR

Name: Christoph Dockhorn
Matrikelnummer: 9883550
Studiengang: Bachelor Informatik
Betreuer: Lihong Ma, Christian Icking

März 2022

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Motivation . . . 5
1.2 Aufgabe . . . 5
1.3 Randbedingungen . . . 6
1.4 Softwareentwicklung . . . 6

2 Stand der Wissenschaft und Technik **9**
2.1 Bestehende Projekte des DLR . . . 9
2.2 Verwendete Technologien . . . 9

3 Anforderungen und Modellierung **13**
3.1 Anforderungen . . . 13
3.2 Modellierung . . . 16

4 Design und Architektur **19**

5 Implementierung **23**
5.1 Arbeitsweise mit Django . . . 23
5.2 Verwendete Dependencies . . . 24
5.3 Modul *groundstations* . . . 25
5.3.1 Erstellung der Tabellen . . . 25
5.3.2 Dateneingabe und Templates . . . 26
5.3.3 Routing . . . 29
5.4 Import/Export-Funktion . . . 30
5.4.1 REST-Schnittstelle . . . 30
5.4.2 Export als Textdatei . . . 35
5.5 Nutzerverwaltung . . . 36
5.6 Logging der Änderungen . . . 37
5.7 Filter . . . 39
5.7.1 Implementierung der Filter . . . 39
5.7.2 Darstellung in der Webanwendung . . . 41
5.7.3 Filtern über die REST-Schnittstelle . . . 42
5.7.4 Filtern der Logs . . . 42
5.8 Erscheinungsbild . . . 43
5.8.1 Allgemeines . . . 43

5.8.2 Cesium 44
5.9 Unittests 44

6 Evaluierung **47**
6.1 Nutzerbefragung 47
6.2 Auswertung 48
6.3 Testabdeckung 49

7 Zusammenfassung und Ausblick **51**

Kapitel 1

Einleitung

1.1 Motivation

Am Deutschen Zentrum für Luft- und Raumfahrt (DLR) fehlt aktuell ein Tool, das aktuelle Informationen über die Verfügbarkeit von Bodenstationen bereitstellt. Bei der Planung neuer Satellitenmissionen sind mehrere Einrichtungen des DLR in unterschiedlichen Entwicklungsphasen beteiligt. Die Concurrent Engineering Facility (CEF) in Bremen konstruiert und entwickelt die Raumfahrzeuge. Das German Space Operations Center (GSOC) in Oberpfaffenhofen betreibt die Bodenstationen und betreut die Missionen, sobald sie im Orbit sind. Um herauszufinden, welche Bodenstationen für die Kommunikation mit einem Satelliten infrage kommen, muss momentan viel telefoniert werden. Die benötigten Daten befinden sich verstreut in verschiedenen Tabellen. Teilweise liegen Daten in unterschiedlichen Formaten vor. Zudem muss händisch geprüft werden, ob die Antennen einer Bodenstation für eine bestimmte Kombination aus Orbit des Satelliten sowie dessen technischen Spezifikationen geeignet sind. In dieser Arbeit soll versucht werden, diese Probleme zu lösen.
Thematisch gliedert sich diese Arbeit in den Bereich der kooperativen Systeme beziehungsweise des kooperativen Arbeitens, da eine Anwendung entwickelt werden soll, die die Zusammenarbeit zwischen verschiedenen Standorten des DLR verbessern soll. Der Begriff Kooperation bedeutet dabei nicht ausschließlich, dass über das Programm ein sozialer Austausch hergestellt werden muss. Kooperation stellt die Zusammenarbeit im engeren Sinne dar. Zur effektiven Zusammenarbeit ist oft eine gemeinsame Datenhaltung mehrerer Akteure erforderlich [16, S. 8]. Diese soll in dieser Arbeit realisiert werden.

1.2 Aufgabe

Im Rahmen dieser Bachelorarbeit soll eine Software erstellt werden, mithilfe derer die verschiedenen Standorte des DLR auf einer gemeinsamen Datengrundlage Entscheidungen im Planungsprozess von Satellitenmissionen treffen können. Um dieses Ziel zu erreichen, wird folgende Vorgehensweise gewählt:

- In Rücksprache mit CEF und GSOC wird eine Anforderungsdefinition erarbeitet.

Dazu gehört eine Auflistung der erforderlichen Anwendungsfunktionalitäten sowie der zu erfassenden Daten der Bodenstationen, siehe Abschnitt 3.1.

- Anschließend wird die Softwarearchitektur geplant und mithilfe von UML modelliert.
- Auf Grundlage dieser Planung wird ein Prototyp erstellt, der die erarbeitete Anforderungsdefinition umsetzt.

1.3 Randbedingungen

Bei der Erfüllung dieser Aufgaben müssen Randbedingungen beachtet werden, die nun erläutert werden.
CEF und GSOC befinden sich an unterschiedlichen Standorten des DLR. Die CEF sitzt in Bremen, während das GSOC in Oberpfaffenhofen bei München angesiedelt ist. Die räumlichen Differenzen werden zusätzlich erschwert, da im DLR teilweise abgetrennte Netze bestehen. Die CEF übernimmt zuweilen auch externe Aufträge, wodurch im Einzelfall eine abgetrennte Netzinfrastruktur eingerichtet wird.
Eine weitere Randbedingung sind die unterschiedlichen Einsatzzwecke, die beide Standorte mit der geplanten Software verbinden. Das GSOC will in dieser Software seine Bodenstationen eintragen und verwalten, während die CEF nur lesend zugreifen muss, um bestimmte Parameter der Bodenstationen abzufragen. Daher sollten die Nutzer der beiden Standorte über unterschiedliche Zugriffsrechte verfügen. Das GSOC muss zur Verwaltung der Stationen über Lese- und Schreibzugriff verfügen, während die CEF nur einen Lesezugriff zum Konsumieren der Daten benötigt.
Bei der Ermittlung der benötigten Daten für die Datenbank ist zu beachten, dass das primäre Ziel der Anwendung ist, der CEF frühzeitig im Planungsprozess Zugriff auf die benötigten Daten der Bodenstationen zu bieten. Zu den einzelnen Antennen der Bodenstationen existieren mehr Daten und Parameter, als von der CEF benötigt werden. Deshalb muss bei der Ermittlung der Anforderungen beachtet werden, dass nur die wesentlichen Daten erfasst werden. Die Daten werden nur in einer bestimmten Phase der Planung von Satellitenmissionen benötigt, um eine Vorauswahl verfügbarer Bodenstationen treffen zu können. Daher ist es nicht erforderlich, dass die Daten in kurzen regelmäßigen Zeitintervallen aktualisiert werden. Sinn und Zweck der Datenbank ist vielmehr, dass die Daten an einem zentralen Ort aktuell gehalten und verwaltet werden, anstatt wie bisher in verschiedenen Excel-Tabellen.

1.4 Softwareentwicklung

In diesem Teilabschnitt wird das Vorgehen bei der Softwareentwicklung beschrieben. Das Vorgehen orientiert sich grob an den Softwareentwicklungsrichtlinien des DLR [29]. Im ersten Schritt werden die Anforderungen an die Software ermittelt. Dazu wird in enger Absprache mit den beteiligten Einrichtungen eine Liste erarbeitet, die die

benötigten Funktionalitäten, Daten und Anwendungsfälle beinhaltet. Es wird dokumentiert und begründet, welche Technologien, Programmiersprachen und Frameworks zum Einsatz kommen sollen. Weiterhin wird die Software zur besseren Planung modelliert. Anschließend wird das geplante Design und die Architektur des verwendeten Frameworks erläutert.
Im zweiten Schritt wird die Software gemäß der vorgegebenen Ziele implementiert. Die Entscheidungen, die bei der Implementierung getroffen werden, werden in einem eigenen Abschnitt erläutert und begründet. In diesem Abschnitt werden zudem Zwischenschritte zur Erfüllung der aufgestellten Ziele beschrieben. Die Softwareentwicklung folgt den Richtlinien der gewählten Programmiersprache. Weiterhin wird eine möglichst modulare Struktur mit einer losen Kopplung der einzelnen Module angestrebt. Die Lesbarkeit und Verständlichkeit des Programmcodes wird durch geeignete Kommentare verbessert. Unnötige Komplexität soll vermieden werden, sodass das Design möglichst einfach und verständlich bleibt. Im Verlauf der Entwicklung werden einzelne Fortschritte in separaten Commits mithilfe eines Versionsverwaltungssystems festgehalten. Dabei wird beachtet, dass nur lauffähige Zwischenstände der Software eingecheckt werden.
Im dritten und letzten Schritt wird die Erfüllung der aufgestellten Ziele getestet. Um die Funktionsfähigkeit der Software zu prüfen werden Unittests verwendet, die die einzelnen Programmbestandteile testen. Weiterhin wird die Software durch Mitarbeiter des DLR begutachtet, um sowohl etwaige Probleme oder fehlerhafte Implementierungen ausfindig zu machen, als auch die Erfüllung der Anforderungen zu beurteilen.

Kapitel 2

Stand der Wissenschaft und Technik

2.1 Bestehende Projekte des DLR

Aktuell existieren zwei Projekte am DLR, die sich mit Bodenstationen befassen. Ein am GSOC entwickeltes Projekt namens GSN_MAP dient der Visualisierung des Netzwerkes von Bodenstationen mithilfe der Plattform Cesium.
Das zweite Projekt heißt Orbitscalc und ist bei Github veröffentlicht [6]. Dieses Programm wurde mit Django entwickelt und ermittelt nach Eingabe von Satellitendaten und einer Auswahl von Antennen, Bodenstationen oder Betreibern die erreichbare Datenübertragungsmenge. Orbitscalc setzt ebenfalls Cesium zur Visualisierung ein.
Beide Programme bieten nur geringe Schnittmengen mit der im Rahmen dieser Arbeit zu entwickelnden Software. Da im Verlauf der Entwicklung die Integration einer Weltkarte erforderlich wurde, wurden beide Programme als Anschauungsmaterial zur Implementierung einer Karte mithilfe von Cesium verwendet. Code wurde dabei nicht übernommen, da die Anforderungen an die Kartenimplementierung im Rahmen dieser Arbeit sehr gering waren, während GSN_MAP und Orbitscalc mehr Features bei der Visualisierung bieten.

2.2 Verwendete Technologien

In diesem Abschnitt wird aufgelistet, welche Technologien und Hilfsmittel zur Erstellung der Arbeit verwendet werden.

- **Entwicklung:** Zur Entwicklung der Anwendung wird das Webentwicklungsframework Django verwendet [9]. Alternativen zu Django wären beispielsweise Node.js für JavaScript [24] oder das PHP-Framework Laravel [26]. Django wurde gewählt, da es für die Programmiersprache Python entwickelt wurde und bereits Erfahrungen bei der Entwicklung mit Python vorhanden sind. Django gilt als schnell, verfügt über eine eingebaute Nutzerverwaltung und erlaubt das

Einbinden von Middleware-Bibliotheken um Probleme wie Sicherheit oder Authentifizierung zu lösen. Zur Datenverwaltung können in Django verschiedene Datenbanksysteme integriert werden, wie etwa MySQL, PostgreSQL oder SQLite. Eine Administrationsoberfläche zur Verwaltung der Datenbankinhalte wird automatisch generiert. Zudem lassen sich die URLs der Webanwendung über reguläre Ausdrücke konfigurieren. Django ist Open Source und frei verfügbar. Darüberhinaus wird Django innerhalb der Forschungsgruppe des DLR, in der diese Arbeit entsteht, häufig bei der Entwicklung von Anwendungen eingesetzt. Da die Anwendung nach Abschluss der Arbeit noch weiterentwickelt werden soll, ist es somit sinnvoll, wenn ein Framework verwendet wird, mit dem auch andere Mitarbeiter bereits Erfahrungen gesammelt haben.

- **Versionsverwaltung:** Zur Versionierung der Software sowie der schriftlichen Ausarbeitung der Arbeit wird Git eingesetzt. Die Quellcodes werden in Gitlab Repositories des DLR hochgeladen.

- **Modellierung:** In dieser Arbeit wird die Modellierungssprache UML eingesetzt [23]. UML ist der Standard im Bereich der Softwaremodellierung. Für die UML-Modellierung wird das kostenlose Tool Dia verwendet.

- **Entwicklungswerkzeuge:** Bei der Softwareentwicklung und dem Verfassen der Arbeit kommt die PyCharm Community Edition zum Einsatz. Diese Entwicklungsumgebung ist frei verfügbar, für die Python-Entwicklung optimiert und kann mithilfe der Erweiterung Texify auch für das Schreiben von LaTeX-Dokumenten verwendet werden.

- **Datenbank:** Django unterstützt die Verwendung verschiedener Datenbanken. In diesem Projekt wird eine PostgreSQL Datenbank eingesetzt [32]. PostgreSQL ist eine objektrelationale Open Source Datenbank, basierend auf SQL. PostgreSQL eignet sich in besonderem Maße für den Einsatz in Django-Anwendungen, da viele Features von Django unterstützt werden. Unter anderem existieren Datentypen in Django, die nur mit PostgreSQL kompatibel sind [11].

- **Test/Evaluierung:** Zum Testen der Funktionsfähigkeit der entwickelten Software werden Unittests eingesetzt. Unittests sind hilfreich, um das richtige Funktionieren von einzelnen Funktionen und Modulen sicherzustellen. Weiterhin werden damit die einzelnen Endpunkte der Webanwendung getestet. Im Rahmen von Django wird für das Erstellen der Tests nicht das Python Modul unittest verwendet [28], sondern die Testklasse django.test.TestCase [8]. Diese Django-spezifische Testklasse ist extra dazu ausgelegt, mit der Datenbank interagieren zu können, weshalb sie für Unittests in Django-Anwendungen besser geeignet ist. Allerdings ist diese Testklasse langsamer als normale Unittests, da jeder Aufruf ein Request-Objekt erzeugt, das alle Middleware-Schichten durchläuft, eine Response erzeugt, welche dann ebenfalls alle Schichten durchläuft [17]. Der Umfang des Projektes ist jedoch überschaubar, entsprechend kann dieser Nachteil vernachlässigt werden, da die Vorteile überwiegen.

Zum Testen von Schnittstellen wurde die kostenfreie Software Postman eingesetzt [27]. Postman ermöglicht das Absenden verschiedener HTTP-Requests, sodass auch POST- und PUT-Requests mit angehängtem JSON getestet werden können.

Kapitel 3

Anforderungen und Modellierung

In diesem Abschnitt wird erläutert welche Anforderungen ermittelt wurden und welchem Zweck diese dienen, wobei die in Abschnitt 1.3 erläuterten Randbedingungen berücksichtigt werden. Weiterhin wird aus den Anforderungen die Modellierung der Software abgeleitet.

3.1 Anforderungen

In Absprache mit CEF und GSOC wurde eine Liste mit Daten ermittelt, die für den Zweck der Datenbank relevant sind. Daher wurden Daten ausgewählt, die hilfreich sind, um die Eignung einer Bodenstation für den Kontakt und Datenaustausch mit einem Satelliten mit einer bestimmten technischen Konfiguration zu beurteilen. Folgende Liste wurde erstellt:

- Standort: Der Standort ist meist unveränderlich, allerdings existieren auch mobile Bodenstationen. Daher muss festgehalten werden, ob der Standort veränderbar ist.
- Haupt- und Backupansprechpartner der Station: Zu jeder Bodenstation sollen Ansprechpartner eingetragen werden können.
- Azimut/Elevation: Azimut und Elevation bestimmen die Richtung zum Satelliten in einem Ost/Nord/Zenit-Koordinatensystem. Beide Werte werden benötigt, um zu ermitteln, wann ein Satellit weit genug über den Horizont gestiegen ist, um Kontakt mit der Bodenstation aufzunehmen [20, S. 67].
- Art des Antennenbandes: Das Antennenband determiniert den Frequenzbereich der Kommunikation. Es folgt eine unvollständige Übersicht der gebräuchlichsten Bandtypen sowie ihrer Frequenzbereiche [34, S. 109]:
  - L-Band: 1,215-1,850 MHz
  - S-Band: 2,025-2,400 MHz
  - C-Band: 3,400-6,725 MHz

  - X-Band: 7,025-8,500 MHz
  - Ku-Band: 10,700-14,500 MHz
  - Ka-Band: 18,000-35,000 MHz
  - V-Band: 37,500-50,200 MHz

- Antennendurchmesser: Der Antennendurchmesser wird in Metern angegeben. Vom Antennendurchmesser hängen viele andere Eigenschaften und Einsatzmöglichkeiten der Antenne ab.

- Unterstützte Frequenzen der Antenne: Diese sind abhängig vom vorhandenen Antennenband. Für Uplink (Versendung eines Signals von der Bodenstation zum Satelliten) und Downlink (Satellit zur Bodenstation) sind jeweils eigene Frequenzbereiche vorgesehen.

- Polarisation: Die Polarisation einer elektromagnetischen Welle wird durch die Richtung der elektrischen Feldkomponente charakterisiert. Bei zirkularer Polarisation wird abhängig von der Richtung der Feldrotation zwischen *right hand circularly polarized* (RHCP) und *left hand circularly polarized* (LHCP) unterschieden [20, S. 385]. Weiterhin kann das elektrische Feld auch horizontal oder vertikal zur Erdoberfläche verlaufen.

- G/T: *Gain-to-noise temperature*, dieser Wert bestimmt die Empfangsempfindlichkeit der Antenne [14, S. 109].

- EIRP: Die *equivalent isotropic radiated power* ist ein Wert, der die Sendeleistung einer Antenne beschreibt [14, S. 109].

- Eine Antenne kann verschiedene Services anbieten. Beispiele sind:

  - Ranging: Wird zur Orbitbestimmung genutzt. Ranging basiert auf der Übertragung einer Tonsequenz an das Raumfahrzeug, welches diese empfängt und über sein Downlinksignal zurücksendet. So kann der radiale Abstand zwischen Raumfahrzeug und Bodenstation bestimmt werden [34, S. 386-387].
  - Doppler: Wird zur Orbitbestimmung genutzt. Dabei wird die Differenz zwischen Uplink- und Downlinkfrequenz bestimmt, um die radiale Geschwindigkeit des Raumfahrzeugs in Bezug auf die Bodenstation mithilfe der Dopplerverschiebung der Frequenz zu bestimmen [34, S. 387].

- Modulationsverfahren: Modulation dient der Frequenzanpassung eines Signals zum Zwecke der Übertragung über Kabel oder Luft.

- Codingverfahren: Beschreibt ebenfalls ein Verfahren zur Signalverarbeitung.

- Gain: Mit dem Wert Gain wird die Richtwirkung und der Wirkungsgrad der Antenne zusammengefasst [20, S. 384]. Er wird durch Durchmesser und Antennentyp determiniert [20, S. 490].

- Mount: Beschreibt die Art und Weise wie die Antenne bewegt werden kann.
- Telemetry/Tracking Combiner: Combiner verbinden Signale, mit dem Ziel, das Signal mit der besten Qualität zu verwenden, oder durch die Kombination beider Signale ein besseres Signal zu rekonstruieren. Combiner werden verwendet, um Redundanz entweder beim Signal oder den Kommunikationsgeräten zu erzeugen, um Atmosphäreneffekten oder der technischen Unvollkommenheit der Systeme entgegenzuwirken. Combiner können je nach Einsatzzweck als pre- oder post-combiner verwendet werden. Telemetry Combiner verbinden Telemetriesignale, während Tracking Combiner für das Verbinden von Trackingsignalen genutzt werden [20, S. 492].

Neben den zu erfassenden Daten wurde auch der gewünschte Funktionsumfang der Anwendung umrissen. Folgende Anforderungen werden an die Anwendung gestellt:

- Einzelne Werte sollen als änderbar bzw. nicht änderbar gekennzeichnet werden, denn nicht alle Werte sind fest und könnten für eine Mission auch verändert werden. Da Missionen langfristig geplant werden, werden dafür auch unter Umständen Investitionen in Upgrades der Bodenstationen durchgeführt, wodurch die Bodenstation den Bedürfnissen der Mission angepasst werden könnte.
- Änderungen an der Datenbank sollen nur per User-Login möglich sein.
- Erfolgte Änderungen sollen nachvollziehbar sein, also muss ein Änderungslog implementiert werden.
- Es sollen verschiedene Nutzergruppen angelegt werden können, da die CEF nur lesenden Zugriff benötigt, während im GSOC auch Schreibrechte gebraucht werden.
- Die Datenbank soll über Import/Export-Funktionen verfügen, da im DLR teilweise abgetrennte Netze bestehen und noch nicht endgültig geklärt ist, wo die Anwendung laufen soll. Daher soll im ungünstigsten Fall auch der Import und Export der Daten möglich sein. Weiterhin soll dadurch die Kopplung anderer Anwendungen ermöglicht werden.
- Ein Export in Textfiles soll möglich sein, da die CEF teilweise mit Textfiles arbeitet, um die Daten in weiteren Tools und Programmen zu nutzen. Zukünftig könnte dort auch eine Anbindung an andere Tools erfolgen, für den Prototyp ist vorerst nur ein Export in Textfiles vorgesehen.

Um die geplante Nutzung der Anwendung zu umreißen wurden folgende Anwendungsfälle formuliert:

- Anzeigen der kompletten Liste: Die komplette Listen der enthaltenen Bodenstationen und Antennen soll angezeigt werden können.

- Filtern nach bestimmten Werten: Mithilfe der Anwendung sollen passende Bodenstationen für Satellitenmissionen gesucht werden können. Es muss daher möglich sein, technische Spezifikationen einzugeben, um die Liste der Bodenstationen entsprechend zu filtern. Für die Auswahl einer passenden Bodenstation sind der Orbit sowie die technische Kompatibilität des Satelliten ausschlaggebend [14, S. 107 ff.].

- Import/Export der Daten: Da aufgrund der Problematik der abgetrennten Netze innerhalb des DLR noch Unklarheit darüber besteht, ob der Zugriff aller Standorte auf eine einzige Datenbank möglich sein wird, soll eine Möglichkeit zum Import und Export der Daten bestehen.

3.2 Modellierung

Im vorherigen Abschnitt wurden die Anforderungen aufgelistet. Aus diesen werden in diesem Teilabschnitt Modelle gebildet, um die geplante Software zu veranschaulichen. Zunächst werden die Anwendungsfälle modelliert. Das Anwendungsfalldiagramm ist in Abbildung 3.1 zu sehen. Es enthält die zuvor genannten Anwendungsfälle.
Der Anwendungsfall „show complete list“ beinhaltet über eine include-Beziehung die Anwendungsfälle „return all groundstations“ und „return all antennas“, da die Listen nur angezeigt werden können, wenn zuvor die entsprechenden Daten abgefragt werden können.
Der Anwendungsfall „filter list“ erfordert, dass Filter bestimmt und die gefilterte Liste zurückgeliefert wird, was über include-Beziehungen veranschaulicht wurde.
Der Anwendungsfall „import/export data“ wird mittels extend-Beziehungen mit den Anwendungsfällen „import JSON“, „export as JSON“ und „export as textfile“ erweitert. In diesem Fall wurden extend-Beziehungen genutzt, da beim Import/Export die gewünschte Variante gewählt werden kann.
Das Klassendiagramm der Anwendung ist in Abbildung 3.2 zu sehen. Bei der Erstellung des Klassendiagramms wurden die vorgesehenen Datentypen des Django-Frameworks verwendet [7]. Um die Anforderung zu erfüllen, dass einzelne Werte als „änderbar“ bzw. „nicht änderbar“ gekennzeichnet werden können, wurden entsprechende boolean Werte aufgenommen. Um nur notwendige boolean-Werte anzulegen wurde im Vorfeld abgeklärt, welche Werte nie verändert werden, etwa der Durchmesser der Antenne. Daher wurden nicht für alle Werte separate „is changeable“-Felder angelegt. Weiterhin wurden die Variablennamen so gewählt, dass die entsprechende Einheit, in der die Werte erfasst werden, aufgenommen wurde. Eine Bodenstation umfasst in der Regel mehrere Antennen. Diese können unterschiedliche Besitzer und/oder Betreiber haben. Eine Antenne kann zudem über mehrere Antennenbänder verfügen. Theoretisch sind beliebig viele Antennenbänder möglich, in der Praxis kann eine Antenne aktuell maximal über drei Bänder verfügen [18]. Es wurde darauf verzichtet eine maximale Anzahl an Bändern pro Antenne festzulegen, damit mögliche zukünftige Entwicklungen keine Anpassungen erfordern.

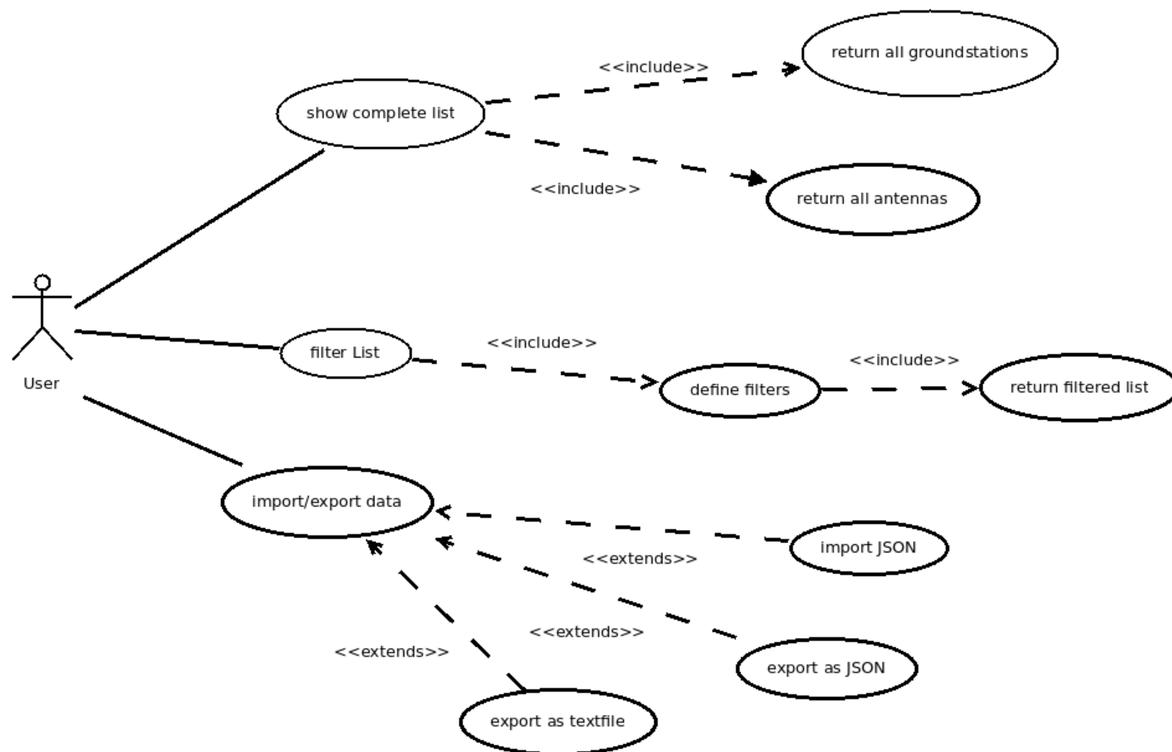


Abbildung 3.1: Anwendungsfalldiagramm

Antennenbänder haben eine bestimmte Frequenzbreite, die vom Typ des Bandes abhängt, siehe Abschnitt 3.1. Allerdings kann sich die Unterteilung der für Uplink und Downlink vorgesehenen Frequenzbereiche bei Bändern gleichen Typs von Antenne zu Antenne unterscheiden. Deshalb wurden die Werte, die die Grenzen der Frequenzbereiche für Uplink und Downlink bestimmen, der Antennenbandklasse zugeteilt.
Eine Antenne kann über eine beliebige Anzahl an Services verfügen. Eine Antenne verfügt über einen einzigen Standort, und an einem bestimmten Standort kann nur eine Antenne stehen, weshalb zwischen Antenna und Coordinates eine 1:1-Beziehung vorliegt.

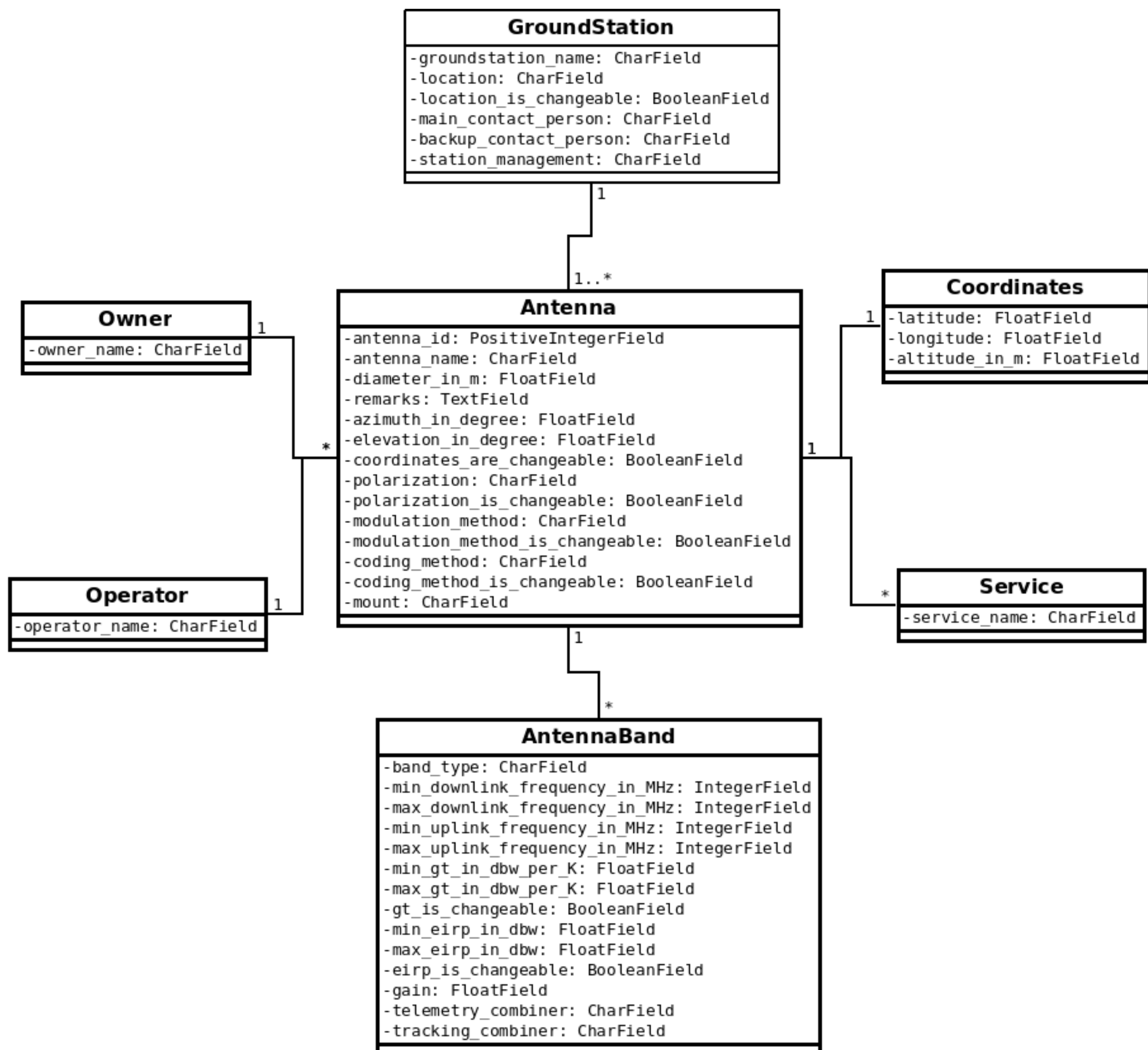


Abbildung 3.2: Klassendiagramm

Kapitel 4

Design und Architektur

In diesem Teilabschnitt wird der Aufbau der Anwendung beschrieben. Das Framework Django gibt einen Großteil der Architektur vor, insofern wird im Folgenden vor allem auf die generelle Architektur von Django-Anwendungen eingegangen.
In vielen Anwendungen wird der Entwurf der Architektur gemäß des Model View Controller (MVC) Musters vorgenommen. Bei diesem Muster übernimmt der Bereich „Model" die Datenhaltung, „View" die Darstellung und „Controller" die Anwendungssteuerung [21]. Das Zusammenspiel dieser drei Bereiche wird in Abbildung 4.1 veranschaulicht. Der Controller steuert die Darstellung in der View sowie die Datenhaltung im Model. Die View kennt ebenfalls das Modell der Datenhaltung, für dessen Darstellung sie zuständig ist, ist selbst aber nicht für die Datenverarbeitung verantwortlich. Die gestrichelten Linien weisen auf indirekte Beziehungen hin, bei denen über Beobachter-Muster das Model die View über Datenänderungen bzw. die View den Controller über zu verarbeitende User-Interaktionen informieren.
Bei Django wird hingegen von einem Model Template View (MTV) Muster gesprochen [13, 15], siehe Abbildung 4.2. Der Bereich „Model" übernimmt die Datenhaltung und die Interaktion mit der Datenbank. Dabei fungiert es als Middleware zwischen Datenbank und View. „Template" ist für die Darstellung der Website auf Client-Seite zuständig und umfasst alles, was im Browser gerendert wird. „View" ist nicht die Django-Version des Controllers, sondern der serverseitige Teil der Darstellung. Die Django-View kommuniziert mittels HTTP-Request und HTTP-Response mit dem Template. Erforderliche Daten werden über das Model beschafft, formatiert und als HTTP-Response an den Client, also das Template, weitergegeben. Abbildung 4.3 verdeutlicht diesen Zusammenhang.
Djangos Projektstruktur unterstützt eine modulare Entwicklung, indem Pakete als Django App angelegt werden. Eine solche App enthält nach der Erstellung standardmäßig folgende Ordner und Dateien:

- migrations-Ordner: Dient der Speicherung aller vorgenommenen Datenbankänderungen.
- __init__.py: Teilt Python mit, dass es sich bei der entsprechenden Django App um ein Package handelt.

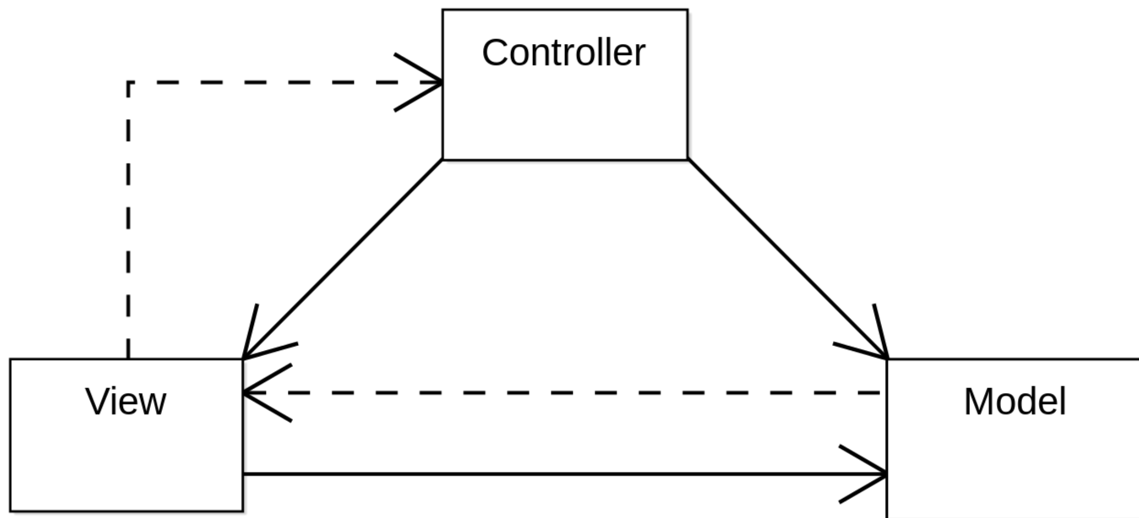


Abbildung 4.1: Model View Controller Diagramm, aus [35]

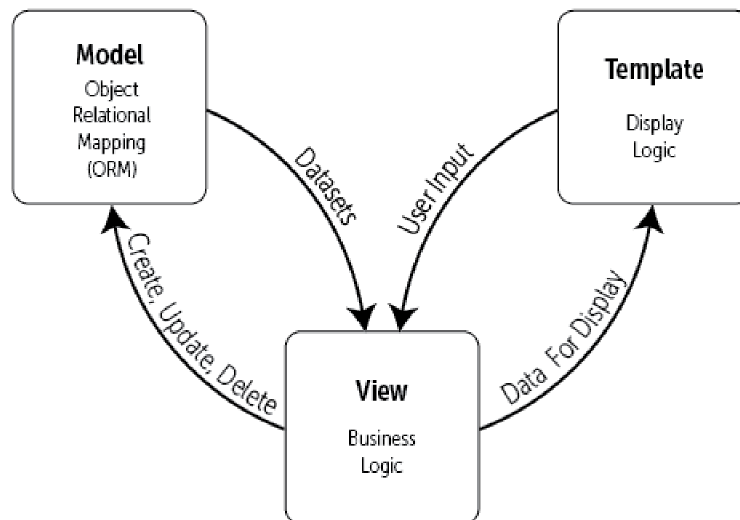


Abbildung 4.2: Model Template View Diagramm von Django, aus [13]

- admin.py: Dient der Registrierung der Datenmodelle in der Django Admin App.
- apps.py: Ist eine Konfigurationsdatei.
- models.py: Enthält die Datenmodelle der App. Hierbei handelt es sich um den Model-Teil der MTV-Architektur.
- tests.py: Enthält Testprozeduren. Es können weitere Testdateien angelegt werden.

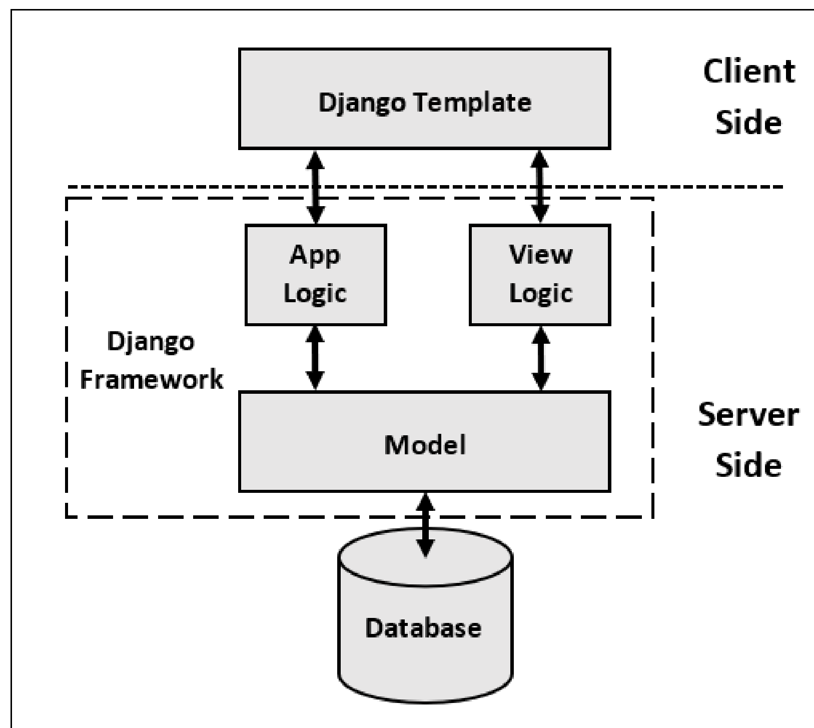


Abbildung 4.3: Die Model Template View Architektur von Django, aus [13]

- views.py: Enthält die Views der App. Entspricht dem View-Teil der MTV-Architektur.

Darüber hinaus können weitere Ordner und Dateien, beispielsweise für HTML-Templates oder Anwendungslogik, angelegt werden.
Für diese Arbeit wurde ein Django-Projekt namens *groundstationDB* angelegt. Darin wurde automatisch eine gleichnamige Django-App angelegt, die die Konfigurationsdateien für die Applikation beinhaltet. Anschließend wurden zwei weitere Django-Apps angelegt: *groundstations* und *webapp*. In *webapp* werden Funktionalitäten der Webanwendung umgesetzt. In *groundstations* sind die Hauptbestandteile der Anwendung implementiert. Neben diesen beiden Django-Apps wurden die Ordner *static* und *templates* im Hauptprojekt angelegt. Auf beide benötigen alle Django-Apps Zugriff, da sie Bilder, Stylesheets und HTML-Templates für das generelle Erscheinungsbild der Webseite enthalten.

Kapitel 5

Implementierung

In diesem Abschnitt wird beschrieben, wie die aufgelisteten Funktionalitäten der Anwendung implementiert wurden. Für die Entwicklung wurden folgende Meilensteine aufgestellt:

- *groundstations*-Modul:
  - Anlegen der Datenmodelle
  - Anzeigen aller Bodenstationen
  - Anzeigen aller Antennen
  - Hinzufügen neuer Antennen und Bodenstationen
- Import-/Exportfunktionen
- Nutzerverwaltung
- Logging aller Datenbankänderungen
- Filtern der Datenbank

5.1 Arbeitsweise mit Django

Ein Webframework wie Django stellt dem Entwickler zahlreiche Hilfsmittel zur Verfügung, um effektiv Webanwendungen entwickeln zu können. Beispielsweise übernimmt Django die Anbindung der Anwendung an die Datenbank. Datenmodelle können in Python angelegt werden, was dem Anlegen von Klassen ähnelt. Diese Modelle werden vom Framework in Tabellen in der Datenbank umgesetzt und verwaltet. Datenbankabfragen müssen dadurch nicht durch SQL-Befehle erfolgen, sondern können komfortabel in Python geschrieben werden.
Weiterhin stellt Django einen Entwicklungsserver bereit, der über die Kommandozeile gestartet wird und sich beim Speichern von Änderungen automatisch aktualisiert.

Django bietet zudem die Möglichkeit, zahlreiche Middleware-Anwendungen einzubinden, um Problembereiche wie die Nutzerauthentifizierung und auch Sicherheitsaspekte anzugehen.
Nach dem Erstellen einer neuen Django Anwendung werden automatisch die wichtigsten Konfigurationsdateien angelegt. Beispielsweise wird in der Datei *settings.py* unter anderem die Middleware eingebunden, der Datenbankzugriff konfiguriert sowie die einzelnen Django-Apps registriert. Weiterhin wird die Datei *urls.py* angelegt, die das Routing übernimmt. Django stellt für das Routing den *urlresolver* bereit. Dieser geht beim Anfragen einer Adresse eine Liste durch und sucht die erste passende Adresse. Die *urls.py*-Datei im Hauptordner wird genutzt, um zu einzelnen Modulen der Webanwendung zu verlinken, die bei Django „Apps“ genannt werden. Jede Django App enthält anschließend eine eigene Datei *urls.py* über die das weitere Routing innerhalb des betreffenden Moduls erfolgt. Die Vereinfachung für den Entwickler besteht darin, dass er lediglich in den entsprechenden Pythondateien Listen anlegen muss, in denen festgehalten wird, welche View-Funktion bei welcher URL aufgerufen wird.
Die Views befinden sich in der Datei *views.py* und bestehen ebenfalls aus Pythoncode, in dem Datenbankabfragen und weitere Geschäftslogik der Anwendung abgehandelt werden um anschließend die benötigten Daten an ein HTML-Template weiterzureichen, das dann im Browser des Nutzers dargestellt wird. Dem Template können neben einfachen Variablen auch Listen, Python-Dictionaries und sogenannte „QuerySets“ übergeben werden. Diese entsprechen Listen, die die Daten einer Datenbankabfrage enthalten. Im Template können sie mithilfe der Django Template Language genutzt werden. Die Django Template Language ermöglicht es dem Entwickler, über Kontrollstrukturen wie *for*-Schleifen oder *if/else*-Anweisungen über die Daten zu iterieren und Bedingungen zu formulieren, sodass die Ansicht in Abhängigkeit von den aktuell vorhandenen Daten dynamisch angepasst werden kann.

5.2 Verwendete Dependencies

Ein Framework wie Django stellt bei der Entwicklung zahlreiche Hilfsmittel zur Verfügung. Um die aufgestellten Anforderungen zu erfüllen, erschien es jedoch sinnvoll, zusätzliche externe Softwarebibliotheken einzubinden, um bestimmte Teilziele besser erreichen zu können. Folgende Dependencies wurden über das Python Paketverwaltungsprogramm *pip* installiert:

- Django REST Framework: Ermöglicht die Erstellung einer REST-Schnittstelle.
- Django-Filter: Vereinfacht das Filtern nach bestimmten Werten sowohl innerhalb der Webanwendung als auch über die REST-Schnittstelle.
- Django-Auditlog: Erleichtert das Logging von Änderungen in der Datenbank.
- Django-Crispy-Forms: Verbessert die Darstellung von Eingabeformularen.
- Coverage.py und django-nose: Helfen bei der Ermittlung der Testabdeckung.

Weitere externe Software wurde über Links im Head des HTML-Dokuments eingebunden:

- Bootstrap: Verbessert das Erscheinungsbild der Webanwendung.
- CesiumJS: Ermöglicht die Darstellung von Geo-Daten auf einer Weltkarte.

Weitere Erläuterungen zu den einzelnen Bibliotheken folgen im jeweiligen Implementierungsschritt.

5.3 Modul *groundstations*

Der erste Meilenstein bei der Entwicklung der Anwendung war die Implementierung des *groundstations*-Moduls. Mithilfe dieses Moduls soll es möglich sein, Bodenstationen, Antennen und weitere benötigte Werte zu erfassen und in der Datenbank zu speichern. Weiterhin soll ermöglicht werden, die erfassten Daten anzuzeigen, zu ändern und zu löschen.

5.3.1 Erstellung der Tabellen

Zuerst wurde die Django-App *groundstations* erstellt. In der darin befindlichen Datei models.py wurden anschließend die benötigten Modellklassen so angelegt, wie sie im Klassenmodell in Abbildung 3.2 zu sehen sind. Die einzelnen Attribute wurden als Felder innerhalb der Modellklassen angelegt. Django bietet die Möglichkeit, *field options* festzulegen, um beispielsweise das Speichern leerer Werte in der Datenbank zu erlauben. Im Folgenden werden die verwendeten *field options* aufgelistet und kurz erläutert:

- *unique*: Standardwert ist False. Wenn die Option auf True gesetzt ist, kann nur ein Objekt der entsprechenden Modellklasse einen bestimmten Wert in diesem Feld haben. Dies wurde bei den Feldern für Name und Ort der Bodenstationen sowie Namen und ID der Antennen verwendet, da diese einzigartig sind und Dopplungen vermieden werden müssen.
- *default*: Legt den Standardwert des Feldes fest. Die Option wurde bei Boolean-Werten verwendet, sowie bei Feldern, die auch leer sein können, sodass dort standardmäßig ein leerer Wert gespeichert wird.
- *null*: Wenn True, dann sind bei diesem Feld auch leere Werte erlaubt. Diese Option wurde bei Feldern benutzt, die leer bleiben können, um das Anlegen von Antennen und Bodenstationen auch mit vorerst unvollständigen Daten zu erlauben.
- *blank*: Wenn True, kann ein Feld leer bleiben. Im Gegensatz zur Option *null* bezieht sich diese Option nicht auf die Speicherung leerer Felder in der Datenbank, sondern auf die Validierung des Feldes im Eingabeformular. Wenn der Wert auf False gesetzt ist, wird beim Versuch ein leeres Feld zu speichern angezeigt, dass ein Eintrag erforderlich ist.

- *choices*: Über diese Option lassen sich einem Textfeld Auswahlmöglichkeiten vorgeben. Diese Option wurde für die Felder „mount“, „telemetry_combiner“, „tracking_combiner“, „band_type“ sowie „polarization“ verwendet, da dort nur bestimmte festgelegte Werte eingetragen werden dürfen.

- *primary_key*: Hiermit kann ein Attribut als Primary Key festgelegt werden, über den das Objekt in der Datenbank referenziert wird. Wenn diese Option genutzt wird sollte ebenfalls *unique* auf True gesetzt werden, um eine Objektinstanz eindeutig identifizierbar zu machen. Standardmäßig erstellt Django für jede Modellklasse, in der kein Attribut als *primary key* festgelegt wurde, einen Primärschlüssel, der über *<objekt>.pk* aufgerufen werden kann. Dabei handelt es sich um einen Integerwert, der bei Null beginnt und mit jeder neu erstellten Objektinstanz inkrementiert wird. In dieser Anwendung wurden für *Antenna* und *GroundStation* eigene Primärschlüssel festgelegt. Die Klasse *Antenna* verfügt über das Attribut *antenna_id*, somit wäre es verwirrend gewesen, durch Django eine weitere ID festlegen zu lassen, die von der nutzerseitig festgelegten ID abweicht. Für Bodenstationen ist keine eigene ID vorgesehen und der Zugriff darauf sollte über den Namen der Bodenstation erfolgen, weshalb *groundstation_name* als Primärschlüssel gewählt wurde. Die Primärschlüssel beider Klassen sind zudem für den Aufbau der URLs erforderlich und sollten für den Nutzer nachvollziehbar sein. Die Schlüssel aller anderen Klassen werden nur zur internen Verwaltung genutzt, insofern war es nicht notwendig dort ebenfalls eigene Primärschlüssel festzulegen.

Weitere Optionen, die verwendet wurden, sind abhängig vom gewählten Feldtyp. Bei Textfeldern etwa kann zusätzlich die maximale Zeichenanzahl festgelegt werden.
Um die im Klassenmodell in Abbildung 3.2 festgelegten 1:1 und 1:n Beziehungen zu realisieren, wurden die Django-Feldtypen *OneToOneField* und *ForeignKey* verwendet. *OneToOneField* wurde für die Beziehung zwischen Antenne und Koordinaten eingesetzt, um die dortige 1:1 Beziehung herzustellen und zu vermeiden, dass ein bestimmter Koordinaten-Wert für mehrere Antennen verwendet werden kann. Bei allen anderen Beziehungen handelt es sich um 1:n-Beziehungen, insofern kam dort *ForeignKey* zum Einsatz. Dieses Feld wird auf der n-Seite der Beziehung gesetzt. Dadurch steht beispielsweise eine Antenne immer in Beziehung zu nur einer Bodenstation. Andersherum können mehrere Antennen die selbe Bodenstation speichern, wodurch einer bestimmten Bodenstation n Antennen zugeordnet werden können. Django legt auf der 1-Seite der Beziehung automatisch ein Set an, über das sich alle Instanzen einer Klasse aufrufen lassen, die einen *ForeignKey* zum Objekt besitzen, also beispielsweise ein *antennaset* bei einer Bodenstation.

5.3.2 Dateneingabe und Templates

Zur Eingabe der Daten wurde die Datei forms.py angelegt. Darin wurden Formularklassen erstellt, in denen festgelegt wurde, welche Attribute bei der Eingabe angezeigt

werden sollen. Generell wurden in allen Klassen alle Attribute der jeweiligen Modellklasse ausgewählt. Eine Ausnahme bilden die Formklassen zu den Modellklassen AntennaBand, Coordinates und Service, bei denen jeweils das Attribut *antenna* von der Anzeige ausgeschlossen wurde. Dies hat den Hintergrund, dass diese Klassen in Bezug zur jeweiligen Antenne stehen und auf der Detailseite der entsprechenden Antenne eingebunden werden. Zudem würde durch die Einbindung des Antennenattributs die Zuordnung zu einer anderen Antenne ermöglicht, was vermieden werden soll.
Die HTML-Templates der Webanwendung wurden im Ordner *templates* innerhalb der *groundstations*-App abgelegt. Im Rahmen dieses Implementierungsschritts wurden Templates für folgende Inhalte erstellt:

- *antenna_list.html*: Zeigt eine Tabelle aller Antennen, mit einem Teil der Daten. Über Buttons können bestehende Antennen bearbeitet oder gelöscht werden. Unterhalb der Tabelle werden die Standorte der Antennen auf einer Weltkarte angezeigt, siehe Abbildung 5.1.

- *groundstation_list.html*: Zeigt eine Tabelle aller Bodenstationen. Über Buttons können bestehende Bodenstationen bearbeitet oder gelöscht werden, zudem können einer Bodenstation weitere Antennen hinzugefügt werden.

- *antenna_detail_form.html*: Zeigt alle Details einer Antenne, mit Eingabefeldern zum Bearbeiten bestehender bzw. Anlegen neuer Antennen. Die Antennenbänder und Services der Antenne werden ebenfalls angezeigt. Über Buttons können weitere Antennenbänder und Services sowie neue Owner oder Operator hinzugefügt werden.

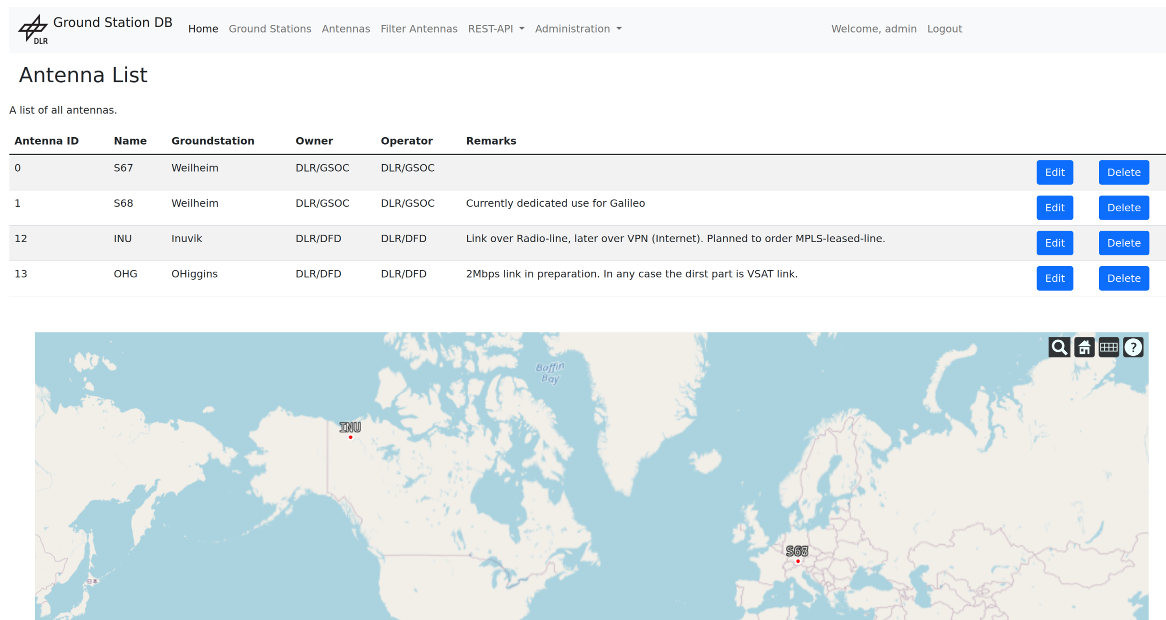
- *groundstation_detail_form.html*: Zeigt alle Details einer Bodenstation mit Eingabefeldern zum Bearbeiten bestehender bzw. Anlegen neuer Bodenstationen, siehe Abbildung 5.2. Bei bestehenden Bodenstationen werden, falls vorhanden, die dazugehörigen Antennen angezeigt.

- *antenna_band_form.html*: Zeigt ein Eingabeformular an, das zum Hinzufügen neuer Antennenbänder zu einer Antenne benötigt wird. Das Bearbeiten bestehender Antennenbänder kann dagegen direkt in den Antennendetails vorgenommen werden.

- *add_operator_form.html*: Zeigt ein Eingabeformular an, das zum Hinzufügen eines neuen Operator zu einer Antenne benötigt wird. Das Bearbeiten bestehender Operator kann direkt in den Antennendetails vorgenommen werden.

- *add_owner_form.html*: Zeigt ein Eingabeformular an, das zum Hinzufügen eines neuen Owner zu einer Antenne benötigt wird. Das Bearbeiten bestehender Owner kann direkt in den Antennendetails vorgenommen werden.

- *add_service_form.html*: Zeigt ein Eingabeformular an, das zum Hinzufügen eines neuen Service zu einer Antenne benötigt wird. Das Bearbeiten bestehender Services kann direkt in den Antennendetails vorgenommen werden.



Antenna ID	Name	Groundstation	Owner	Operator	Remarks		
0	S67	Weilheim	DLR/GSOC	DLR/GSOC		Edit	Delete
1	S68	Weilheim	DLR/GSOC	DLR/GSOC	Currently dedicated use for Galileo	Edit	Delete
12	INU	Inuvik	DLR/DFD	DLR/DFD	Link over Radio-line, later over VPN (Internet). Planned to order MPLS-leased-line.	Edit	Delete
13	OHG	OHiggins	DLR/DFD	DLR/DFD	2Mbps link in preparation. In any case the dirst part is VSAT link.	Edit	Delete

Abbildung 5.1: Screenshot der Antennenliste

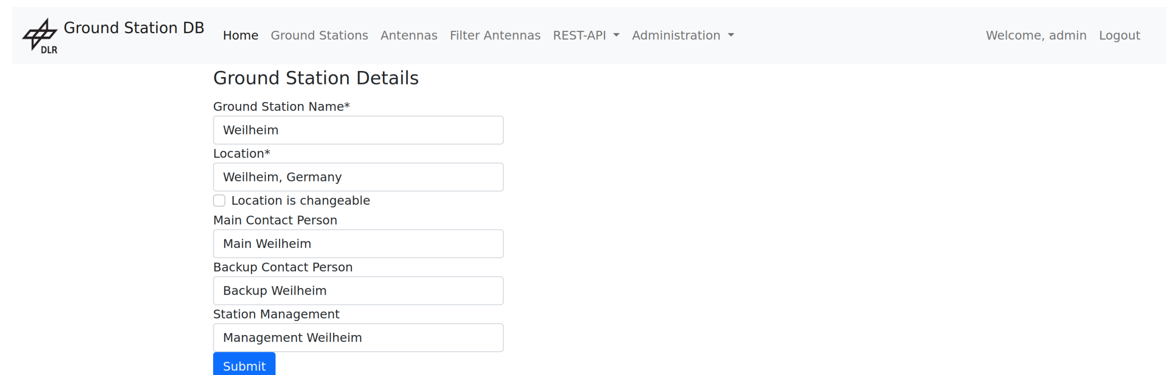


Abbildung 5.2: Screenshot der Detailseite einer Bodenstation

- *delete_antenna.html*: Dieses Template wird beim Löschen einer Antenne aufgerufen. Der Nutzer muss hier bestätigen, ob er die genannte Antenne inklusive aller zugehörigen Antennenbänder, Koordinaten sowie Services löschen möchte. Die Bestandteile werden dazu einzeln aufgelistet.

- *delete_groundstation.html*: Dieses Template dient dem Löschen einer Bodenstation. Der Nutzer muss bestätigen, dass er die genannte Bodenstation samt aller Antennen und deren Bestandteile löschen möchte. Die betroffenen Antennen werden aufgelistet.

5.3.3 Routing

Jede Django-App enthält automatisch die Datei urls.py. In dieser werden die URL-Bezeichnungen sowie die Methoden der View festgelegt, die beim Aufruf der jeweiligen URL zum Einsatz kommen. Folgende URL-Endpunkte wurden im Rahmen dieses Implementierungsschritts erstellt:

- *host:/groundstations/list*: Anzeige der Liste der Bodenstationen.
- *host:/groundstations/groundstation/pk*: Anzeige der Details einer Bodenstation, „pk“ steht für den Primary Key der jeweiligen Bodenstation, der bei der Erstellung von Django vergeben wird.
- *host:/groundstations/groundstation/pk/delete*: Anzeige der Seite zum Löschen der Bodenstation mit dem unter „pk“ spezifizierten Primary Key.
- *host:/groundstations/groundstation/new*: Anlegen einer neuen Bodenstation.
- *host:/groundstations/antennas*: Anzeige der Liste der Antennen.
- *host:/groundstations/antennas/antenna/pk*: Anzeige der Details einer Antenne, „pk“ steht für den Primary Key der jeweiligen Antenne, der bei der Erstellung von Django vergeben wird.
- *host:/groundstations/antennas/antenna/pk/delete*: Anzeige der Seite zum Löschen der Antenne mit dem unter „pk“ spezifizierten Primary Key.
- *host:/groundstations/groundstation/pk/new-antenna*: Anzeige der Seite zum Erstellen einer neuen Antenne. Der Primary Key bezieht sich auf die Bodenstation, da das Erstellen einer neuen Antenne immer im Zusammenhang mit einer bestimmten Bodenstation erfolgt.
- *host:/groundstations/antennas/antenna/pk/band-detail*: Anlegen eines neuen Antennenbandes, zugehörig zur Antenne mit dem entsprechenden Primary Key.
- *host:/groundstations/antennas/antenna/pk/add-service*: Anlegen eines neuen Services, zugehörig zur Antenne mit dem entsprechenden Primary Key.
- *host:/groundstations/antennas/antenna/pk/add-owner*: Anlegen eines neuen Owners, zugehörig zur Antenne mit dem entsprechenden Primary Key.
- *host:/groundstations/antennas/antenna/pk/add-operator*: Anlegen eines neuen Operators, zugehörig zur Antenne mit dem entsprechenden Primary Key.

5.4 Import/Export-Funktion

In diesem Abschnitt wird die Implementierung von Datenimport und -export beschrieben. Um den Datenaustausch mit anderen Anwendungen zu fördern, wurde beschlossen, Import und Export über eine REST-Schnittstelle zu ermöglichen. REST-Schnittstellen eignen sich insbesondere für die Maschine-zu-Maschine-Kommunikation und sind ein Standard beim Datenaustausch in verteilten Systemen. REST Services nutzen das HTTP Protokoll zur Datenübertragung. Daten werden dabei zumeist im XML oder JSON Format versendet. REST ist eine leichtgewichtigere Alternative zum SOAP Protokoll und bietet zusätzlich bessere Skalierbarkeit, Kompatibilität und Performance [33].

5.4.1 REST-Schnittstelle

Zur Implementierung einer REST-Schnittstelle mithilfe von Django muss ein zusätzliches Paket namens *django-rest-framework* installiert werden. Anschließend wurde eine neue Django Applikation namens „api“ erstellt, in der alle Dateien zur Realisierung der REST-Schnittstelle erstellt werden. Die zusätzlichen Dateien hätten auch im Modul *groundstations* erstellt werden können. Darauf wurde jedoch verzichtet um einen möglichst modularen Aufbau der Anwendung zu gewährleisten und die einzelnen Bestandteile besser voneinander zu trennen.
Zur Darstellung der Schnittstelle auf der Homepage werden im Django REST Framework sogenannte ViewSets eingesetzt. In dieser Anwendung sollen sowohl die Antennen als auch die Bodenstationen über eigene Endpunkte verfügen. Daher wurde die Datei *viewsets.py* erstellt. In dieser wurden die Klassen *AntennaViewSet* und *GroundStationViewSet* erstellt. Ein ViewSet benötigt als Attribute lediglich ein QuerySet des darzustellenden Modells, also entweder alle Antennen oder alle Bodenstationen, sowie einen Serialisierer für das Modell. Die Darstellung erfolgt im JSON-Format. Um bei einem POST-Request nicht nur einzelne Objekte, sondern eine Liste an Objekten zu verarbeiten, muss die *create*-Funktion des ViewSets derartig überschrieben werden, dass erkannt wird ob eine Liste übergeben wird um anschließend den Serialisierer entsprechend zu konfigurieren.
Zur Serialisierung der Daten werden *Serializer* für jede Modellklasse benötigt. Die Datei *serializers.py* wurde erstellt, um darin die entsprechenden Serialisiererklassen zu erstellen. Im Gegensatz zu den ViewSets werden Serialisierer für alle Modellklassen benötigt. Die Serialisierer für Antennenbänder, Services und Koordinaten werden benötigt, um die Modellklasse festzulegenen und zu konfigurieren, welche Attribute angezeigt werden sollen. Da Coordinates, AntennaBand sowie Service immer im Zusammenhang mit einer Antenne angezeigt werden, soll die Antenne nicht im entsprechenden Objekt angezeigt werden. Zudem wurde auch die Anzeige des Primary Key ausgeschlossen, da dieser von Django, wenn nicht anders definiert, automatisch für jede Objektinstanz vergeben wird und für den Nutzer meist irrelevant ist. Die Serialisierer im Django REST Framework verfügen automatisch über Funktionen, um Objekte zu speichern, zu aktualisieren, anzuzeigen oder zu löschen. Allerdings gilt dies nicht für eventuell vorhandene verschachtelte Objektinstanzen anderer Klassen. Daher mussten die Funktionen *create*

und *update* der Klassen *AntennaSerializer* und *GroundStationSerializer* überschrieben werden, um auch die Speicherung bzw. Aktualisierung der enthaltenen Objekte, etwa Antennenbänder, korrekt vornehmen zu können.
Bei der Konfiguration der Serialisierer trat ein Problem mit der Anzeige der Attribute der einzelnen Objekte auf. Da Antennen sowohl einzeln als auch als Bestandteil einer Bodenstation angezeigt werden können, musste sich die Darstellung leicht unterscheiden. Wenn die Antenne als verschachtelter Bestandteil einer Bodenstation angezeigt wird, ist das Attribut *groundstation* innerhalb des Antennen-JSON überflüssig. Zudem könnte fehlerhaftes Verhalten auftreten, wenn eine gekoppelte Anwendung ein fehlerhaftes JSON importieren möchte, bei dem eine verschachtelte Antenne eine andere Bodenstation angibt als die, der sie zugeordnet ist. Deshalb wurde zusätzlich die Klasse *NestedAntennaSerializer* erstellt, die für die Darstellung der Antennen im *GroundstationSerializer* genutzt wird. In dieser Klasse wurden nur die entsprechenden Konfigurationen bezüglich der Anzeige der Attributfelder vorgenommen, ein Überschreiben der Funktionen war nicht nötig, da diese Klasse selbst keine Datenspeicherung übernehmen muss.
In Listing 5.1 ist das JSON einer Antenne zu sehen. Im Vergleich dazu kann in Listing 5.2 das JSON einer Bodenstation mit einer Antenne betrachtet werden. Während im JSON der Antenne die Bodenstation mit aufgeführt wird, ist dies nicht notwendig, wenn die Antenne als Bestandteil einer Bodenstation angegeben wird, da ersichtlich ist, zu welcher Bodenstation die Antenne gehört.

```
1      {
2          "antenna_id": 2,
3          "coordinates": {
4              "latitude": 47.8811825778,
5              "longitude": 11.0782031361,
6              "altitude_in_m": 673.418
7          },
8          "antenna_bands": [
9              {
10                 "band_type": "X-Band",
11                 "min_eirp_in_dbw": null,
12                 "max_eirp_in_dbw": null,
13                 "eirp_is_changeable": true,
14                 "min_downlink_frequency_in_MHz": 8400,
15                 "max_downlink_frequency_in_MHz": 8440,
16                 "min_uplink_frequency_in_MHz": null,
17                 "max_uplink_frequency_in_MHz": null,
18                 "min_g_t_in_dbW_per_K": 44.0,
19                 "max_g_t_in_dbW_per_K": 44.0,
20                 "gt_is_changeable": false,
21                 "gain": 25.0,
22                 "telemetry_combiner": "Yes",
23                 "tracking_combiner": "No"
24             }
```

```
        ],
        "services": [
            {
                "service_name": "Doppler"
            },
            {
                "service_name": "Angle Data"
            }
        ],
        "owner": "DLR/GSOC",
        "operator": "DLR/GSOC",
        "groundstation": "Weilheim",
        "antenna_name": "S69",
        "diameter_in_m": 15.0,
        "remarks": "Currently dedicated use for Galileo",
        "mount": "Az/El",
        "azimuth_in_degree": 15.0,
        "elevation_in_degree": 15.0,
        "polarization": "RHCP & LHCP",
        "polarization_is_changeable": true,
        "modulation_method": "modulation",
        "modulation_method_is_changeable": true,
        "coding_method": "coding",
        "coding_method_is_changeable": true,
        "coordinates_are_changeable": false
    }
```

Listing 5.1: Beispiel-JSON einer einzelnen Antenne

```
    {
    "groundstation_name": "Inuvik",
    "antennas": [
        {
            "antenna_id": 12,
            "coordinates": {
                "latitude": 68.3183,
                "longitude": -133.544,
                "altitude_in_m": 96.0
            },
            "antenna_bands": [
                {
                    "band_type": "S-Band",
                    "min_eirp_in_dbw": 70.0,
                    "max_eirp_in_dbw": 70.0,
                    "eirp_is_changeable": true,
```

```
17                     "min_downlink_frequency_in_MHz": 2200,
18                     "max_downlink_frequency_in_MHz": 2300,
19                     "min_uplink_frequency_in_MHz": 2025,
20                     "max_uplink_frequency_in_MHz": 2120,
21                     "min_g_t_in_dbW_per_K": 22.5,
22                     "max_g_t_in_dbW_per_K": 22.5,
23                     "gt_is_changeable": false,
24                     "gain": 5.0,
25                     "telemetry_combiner": "Yes",
26                     "tracking_combiner": "No"
27                 },
28                 {
29                     "band_type": "X-Band",
30                     "min_eirp_in_dbw": 70.0,
31                     "max_eirp_in_dbw": 70.0,
32                     "eirp_is_changeable": true,
33                     "min_downlink_frequency_in_MHz": 7600,
34                     "max_downlink_frequency_in_MHz": 8500,
35                     "min_uplink_frequency_in_MHz": null,
36                     "max_uplink_frequency_in_MHz": null,
37                     "min_g_t_in_dbW_per_K": 36.0,
38                     "max_g_t_in_dbW_per_K": 36.0,
39                     "gt_is_changeable": false,
40                     "gain": 25.0,
41                     "telemetry_combiner": "Yes",
42                     "tracking_combiner": "No"
43                 }
44             ],
45             "services": [
46                 {
47                     "service_name": "Doppler"
48                 },
49                 {
50                     "service_name": "Angle Data"
51                 }
52             ],
53             "owner": "DLR/DFD",
54             "operator": "DLR/DFD",
55             "antenna_name": "INU",
56             "diameter_in_m": 13.0,
57             "remarks": "Link over Radio-line, later over
                  VPN (Internet). Planned to order MPLS-leased
                  -line.",
58             "mount": "Az/El",
```

```
59                "azimuth_in_degree": 5.0,
60                "elevation_in_degree": 5.0,
61                "polarization": "RHCP & LHCP",
62                "polarization_is_changeable": true,
63                "modulation_method": "modulation",
64                "modulation_method_is_changeable": true,
65                "coding_method": "coding",
66                "coding_method_is_changeable": true,
67                "coordinates_are_changeable": false
68           }
69       ],
70       "location": "Inuvik, Canada",
71       "location_is_changeable": false,
72       "main_contact_person": "Main contact inuvik",
73       "backup_contact_person": "backup inuvik",
74       "station_management": "management inuvik"
75   }
```

Listing 5.2: Beispiel JSON einer Bodenstation mit einer Antenne

Das Routing erfolgt wie gehabt in der Datei *urls.py* innerhalb der Django Applikation. Die akzeptierten HTTP-Methoden unterscheiden sich dabei je nach Endpunkt. Folgende HTTP-Methoden werden am Häufigsten benötigt [31, S . 562]:

- HEAD: Fordert den Header des Dokuments an.
- GET: Fordert ein Dokument an, wird zur Datenabfrage genutzt.
- PUT: Anfrage um ein Dokument zu speichern, etwa bei einer Aktualisierung.
- POST: Anfrage, um ein neues Dokument einer Sammlung hinzuzufügen. Wird zur Erstellung neuer Instanzen genutzt.
- DELETE: Anfrage um ein Dokument zu löschen.

Über allgemeine Endpunkte werden alle Antennen bzw. Bodenstationen angezeigt. Diese Listenansicht akzeptiert neben GET-Requests für die Datenabfrage auch POST-Requests zum Einfügen neuer Antennen. An die allgemeine URL kann, per Slash getrennt, eine ID angehängt werden, um nur eine bestimmte Instanz einer Antenne bzw. Bodenstation zu erhalten. Die Instanzansicht erlaubt zusätzlich PUT-Requests zum Aktualisieren der betreffenden Instanz, sowie DELETE-Requests um eine Instanz zu löschen.
Folgende Endpunkte wurden für die REST-Schnittstelle festgelegt:

- *host:/api/antennas/*: Über diesen Endpunkt wird das *AntennaViewSet* aufgerufen, das die komplette Liste der Antennen im JSON Format anzeigt.

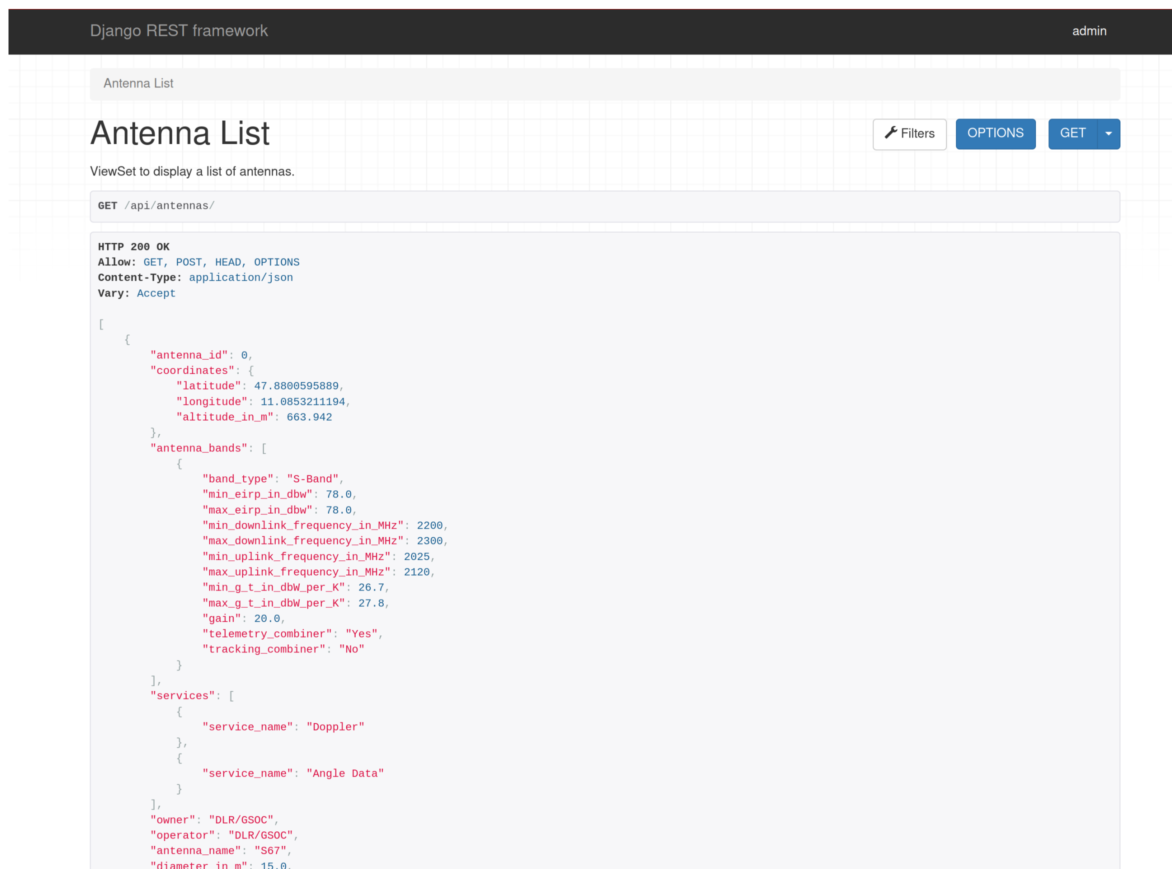


Abbildung 5.3: Screenshot des Antennen-Endpunktes der REST-Schnittstelle

- *host:/api/groundstations/*: Über diesen Endpunkt wird das *GroundStationViewSet* aufgerufen, das die komplette Liste der Bodenstationen im JSON Format anzeigt.

Die Darstellung erfolgt in beiden Endpunkten über eine HTML-Seite, die vom Django REST Framework automatisch erzeugt wird, siehe Abbildung 5.3. Diese Seite stellt zusätzlich ein Eingabefeld zum Hinzufügen neuer Instanzen mittels POST-Request zur Verfügung. Wenn über die Schnittstelle eine andere Anwendung gekoppelt werden soll, braucht man allerdings die Möglichkeit, nur die reinen JSON-Daten abzufragen. Dies erfolgt über eine Formatfestlegung in der URL. Beispielsweise liefert die URL *host:/api/antennas/?format=json* alle in der Datenbank enthaltenen Antennen als JSON-String zurück.

5.4.2 Export als Textdatei

Die CEF hatte als zusätzliche Anforderung zum Datenexport die Möglichkeit des Exports als Textdatei genannt. Nachdem die REST-Schnittstelle implementiert war, sollte zunächst geklärt werden, ob der Datenexport über die Schnittstelle für die Zwecke der CEF ausreichen würde. Zusätzlich hätte besprochen werden müssen, in welchem Format die Daten in der Textdatei hätten dargestellt werden müssen. Allerdings konnte

kein Mitarbeiter der CEF die Zeit dafür erübrigen, sodass diese Abstimmung nicht stattfinden konnte. Da Unklarheiten bezüglich des Formats und damit generell bezüglich dem Nutzen des Features bestanden, wurde mit der Betreuung vereinbart, dass diese Anforderung im Rahmen der Bachelorarbeit anders umgesetzt werden sollte, indem die Daten als JSON exportiert werden.
Der Dateiexport wurde im Bereich „Filter Antennas“ implementiert. Es werden alle momentan in der Ergebnistabelle enthaltenen Antennen in eine .json-Datei exportiert und heruntergeladen. Dadurch können gefilterte Datensätze exportiert werden. Zur Serialisierung der Daten wurde der *AntennaSerializer* aus Abschnitt 5.4.1 verwendet, dem zur Serialisierung das jeweils aktuelle Queryset des *AntennaFilter* übergeben wird. Mit dieser Lösung wurde gewährleistet, dass ein Dateiexport möglich ist, worauf aufgrund der Möglichkeit abgetrennter Netzwerke innerhalb des DLR nicht verzichtet werden sollte.

5.5 Nutzerverwaltung

Eine Anforderung an die Anwendung war, dass Nutzer unterschiedliche Freigaben erhalten können sollten. Die CEF benötigt beispielsweise nur Lesezugriff, während das GSOC auch Schreibzugriff benötigt, um die Daten zu verwalten. Django stellt standardmäßig Mittel zur Nutzerverwaltung zur Verfügung, die für die hier genannten Zwecke ausreichen. So ist es ohne große Änderungen möglich, Nutzer anzulegen und in Nutzergruppen mit unterschiedlichen Freigaben einzuteilen. Django legt automatisch Permissionklassen für jede in der Datenbank vorhandene Modellklasse an. Diese umfassen die Methoden *view*, *add*, *change* und *delete*. Es können darüberhinaus benutzerdefinierte Permissions für einzelne Modellklassen festgelegt werden, was im Rahmen dieser Anwendung jedoch nicht notwendig war.
Für Programmbestandteile, die das generelle Funktionieren der Webanwendung betreffen, wurde die Django-Applikation *webapp* erstellt. Darin wurden View-Funktionen und Templates angelegt, um neue Nutzer zu registrieren und bestehende Nutzer einzuloggen. Die dazu benötigten Funktionen sind im Standard-Usermodell von Django bereits enthalten und mussten nur entsprechend angewendet werden. Anschließend wurden über die Django Adminseite zwei Nutzergruppen angelegt: Eine Gruppe mit Leseberechtigungen für alle Modellklassen der *groundstations*-App und eine weitere Gruppe, die darüber hinaus noch über Berechtigungen zum Erstellen, Bearbeiten und Löschen dieser Klassen verfügt. Diese Konfiguration sollte beim Aufsetzen der Anwendung jeweils vom Administrator vorgenommen werden, um das Funktionieren der einzelnen Programmbestandteile zu gewährleisten und die Nutzerverwaltung zu erleichtern.
Wenn ein neuer Nutzer registriert wurde verfügt er zunächst über keine Berechtigungen. Er muss zunächst von einem Admin die entsprechenden Freigaben bekommen, indem er entweder einer der beiden Nutzergruppen zugewiesen wird, oder ihm im einzelnen die entsprechenden Permissions gewährt werden. Mit Leseberechtigungen kann der Nutzer nun die Listen und Details der Bodenstationen und Antennen sehen, die Antennen filtern und das Änderungslog sehen. Um die Daten zu Ändern, zu Löschen oder zu Erstellen benötigt der Nutzer jedoch auch Schreibzugriff.

Um die Abfrage der Permissions durchzuführen wurden alle View-Funktionen des *groundstations*-Moduls derart angepasst, das vor dem Anzeigen einer Liste oder der Übernahme von Änderungen zunächst geprüft wird, ob die entsprechenden Permissions vorhanden sind. Ist dies nicht der Fall, wird eine Seite aufgerufen, die den Status *403 Permission Denied* anzeigt. Kehrt der Nutzer zur vorherigen Seite zurück wird ihm zusätzlich die Information angezeigt, dass er nicht über die erforderlichen Freigaben verfügt und sich an einen Administrator wenden sollte.
Neben der Implementierung der Nutzerauthentifizierung in der Webanwendung muss auch die REST-Schnittstelle über Möglichkeiten verfügen, den Nutzer zu authentifizieren und seine Berechtigungen zu prüfen. Dafür existieren verschiedene Möglichkeiten, indem in der Datei *settings.py* der Anwendung unter *REST_FRAMEWORK* festgelegt wird, welche Authentifizierung zum Einsatz kommen soll und welche Freigabeklasse verwendet wird. Zur Authentifizierung wurden die Klassen *TokenAuthentication* und *SessionAuthentication* aus dem *rest_framework*-Paket ausgewählt. *SessionAuthentication* ermöglicht es einem eingeloggten Nutzer mit den entsprechenden Freigaben, über die Anwendung auf die Webansicht der Schnittstelle zuzugreifen und dort Änderungen und Abfragen vorzunehmen.
Diese Authentifizierungsvariante ist allerdings unpraktikabel, wenn Anwendungen gekoppelt werden müssen. Dazu wurde die *TokenAuthentication* implementiert. Dabei muss zunächst ein Token abgefragt werden, indem ein POST-Request an die dafür erstellte URL *host:/api/api-token-auth/* gesendet wird, der als Formdaten den Nutzernamen und das Passwort enthält. Sind die Daten valide, erhält der Client in der Response das Token als String. Dieses Token muss bei allen weiteren HTTP-Requests im Header unter *Authorization* mitgesendet werden, um den Nutzer zu authentifizieren.
Um die Berechtigungen des Nutzers zu prüfen wurde im *api*-Modul die Datei *permissions.py* angelegt, in der die Klasse *APIPermission* erstellt wurde. Diese Klasse wurde als Attribut den ViewSets übergeben. Sie prüft bei GET-Requests ob der Nutzer authentifiziert ist und über alle Leseberechtigungen verfügt, und im Falle aller anderen Requestarten, ob er darüber hinaus auch alle Schreibrechte besitzt. Dadurch wird verhindert, dass Nutzer ohne Schreibzugriff über die Schnittstelle Daten ändern können.

5.6 Logging der Änderungen

Eine Anforderung betrifft das Logging der Änderungen an der Datenbank. Damit jederzeit nachvollziehbar ist, welche Änderungen von welchem Nutzer vollzogen wurden, ist es erforderlich alle Änderungen zu speichern. Hierzu wurde auf die externe Lösung „django-auditlog“ zurückgegriffen [19]. Diese App kann über den Python Package Installer installiert und anschließend in der Datei *settings.py* in das Projekt eingebunden werden. Weiterhin wird über das Paket eine zusätzliche Middleware bereitgestellt, durch deren Einbindung der Nutzer geloggt werden kann, der die aktuelle Änderung durchgeführt hat.
Anschließend müssen alle Modelle, deren Änderungen aufgezeichnet werden sollen, in Auditlog registriert werden. Das erfolgt am Ende der jeweiligen Datei *models.py* indem beispielsweise die Zeile „auditlog.register(GroundStation)“ hinzugefügt wird.

Dies wurde für die sieben Modellklassen durchgeführt, die für die Datenhaltung relevant sind. Auditlog legt anschließend bei jeder Änderung „Log Entry“-Objekte an, die in der Datenbank in einer eigenen Relation gespeichert werden und wie alle anderen Objekte gesucht, gefiltert und aufgerufen werden können.
Zur Auflistung der Log-Änderungen wurden eine View-Funktion und ein HTML-Template innerhalb des *webapp*-Moduls erstellt, da das Logging eine Funktionalität der Webanwendung ist, auch wenn aktuell nur Änderungen von Modellen aus dem *groundstations*-Modul geloggt werden. Die View-Funktion prüft zunächst die Leseberechtigungen des Nutzers. Sind diese vorhanden wird über alle Logeinträge iteriert und dabei eine Liste mit Daten befüllt, die für die Darstellung im Template erforderlich sind. Im Template wird anschließend eine Tabelle gerendert, die nach dem Zeitpunkt der Änderungen sortiert ist, wobei die aktuellste Änderung zuerst angezeigt wird. Da die Liste der Änderungen sehr lang werden kann, und bereits bei moderater Länge eine kurze Ladezeit spürbar wurde, wurde eine seitenweise Darstellung der Tabelle gewählt, wobei eine Seite maximal zehn Logeinträge beinhaltet. Folgende Informationen werden in der Tabelle angezeigt:

- Date: Zeitpunkt der Änderung.
- User: Nutzer, der die Änderung vollzogen hat.
- Object: Objekt, das geändert wurde.
- Model: Modellklasse des geänderten Objekts.
- Action: Die durchgeführte Änderungsaktion. Möglichkeiten sind CREATE, UPDATE oder DELETE.
- Field: Eine Auflistung aller geänderten Felder des Objektes.
- From: Der alte Wert des geänderten Feldes.
- To: Der neue Wert des geänderten Feldes.

Bei der Implementierung des Loggings gab es zunächst ein Problem beim Logging aller Änderungen, die über die REST-Schnittstelle erfolgten. Dort wurden zwar die einzelnen Änderungseinträge richtig geloggt, allerdings wurde beim Nutzernamen jeweils „None“ angezeigt, weshalb nicht mehr nachvollziehbar war, wer die Änderungen durchgeführt hat. Eine Recherche ergab, dass das Problem mit Auditlog selbst zusammenhing. Die Ursache war, dass Auditlog erwartet, dass der User auf der Middlewareebene geloggt wird. Das Django REST Framework führt die Nutzerauthentifizierung jedoch auf der Viewebene durch, also bevor der Request verarbeitet und die Response erzeugt wird. Deshalb kann Auditlog den Nutzer nicht auf der Middlewareebene mit dem Änderungsereignis verbinden.
Da es sich hierbei um ein generelles Problem von Auditlog handelte, konnte auf bestehende Lösungsvorschläge und Workarounds zurückgegriffen werden. Die gewählte Lösung basiert zu großen Teilen auf einer vorgeschlagenen Variante eines Github-Nutzers [1]

und umfasst die Erstellung eines Mixin. Mixins werden genutzt, um einer Klasse zusätzliche Funktionalitäten zur Verfügung zu stellen bzw. um ein bestimmtes Feature für verschiedene Klassen verfügbar zu machen. Das Mixin wurde in der Datei *mixins.py* innerhalb des *api*-Moduls angelegt. Der Code der Vorlage wurde für ältere Versionen von Django und Auditlog geschrieben und musste deshalb an einigen Stellen angepasst werden. Anschließend musste das erstellte Mixin als Parameter den Viewsets übergeben werden, um das Logging der Nutzer auch über die Schnittstelle zu ermöglichen.

5.7 Filter

Eine elementare Anforderung an die Anwendung ist die Implementierung einer Möglichkeit, die Antennen nach bestimmten Daten zu Filtern. Dadurch soll die Suche nach passenden Antennen im Planungsprozess von Satellitenmissionen vereinfacht werden. Da sich das Interesse hierbei auf die Antennen beschränkt, wurde lediglich eine Filtermöglichkeit für Antennen und ihre zugehörigen Elemente wie Antennenbänder und vorhandene Services implementiert. Auf die Entwicklung eines Filters für Bodenstationen wurde mangels Bedarf verzichtet.

5.7.1 Implementierung der Filter

Das Filtern soll sowohl über die Webanwendung durch Ausfüllen eines Formulars möglich sein, als auch über die Schnittstelle, sodass gekoppelte Anwendungen nach spezifischen Parametern suchen können. Zur Implementierung wurde auf die externe Django-App „django-filter“ zurückgegriffen [12]. Mithilfe von django-filter lassen sich mit wenig Aufwand Filter sowohl für die Webanwendung als auch für REST-Schnittstellen erstellen. Dazu muss jeweils eine eigene Filtersetklasse erstellt werden, die im jeweiligen Modul in der Datei *filter.py* platziert wurde. Die Inhalte der Filtersetklassen sind nahezu identisch. Allerdings erbt die Filtersetklasse für die Webanwendung von *django_filters.FilterSet*, während die Filtersetklasse für die Schnittstelle von *django_filters.rest_framework.FilterSet* erbt. Die Einbindung der Filter in den Modulen wird gesondert im jeweiligen Unterabschnitt betrachtet. Zunächst wird erläutert, wie die generelle Erstellung der Filtersetklassen funktioniert, und welche Entscheidungen dabei getroffen wurden.
Wie bei der Erstellung von Formklassen kann festgelegt werden, auf welches Modell sich die Filtersetklasse bezieht, und welche Felder des Modells beinhaltet und ausgeschlossen werden sollen. In diesem Fall wurde demnach das Modell *Antenna* gewählt. Das Feld *remarks* wurde ausgeschlossen, da dies ein freies Textfeld mit Platz für Notizen zur jeweiligen Antenne ist, wo ein Filter nicht notwendig ist. Außerdem wurde kein Filter für die Antennen-ID implementiert, da kein Anwendungsfall existiert, bei dem ein Filtern nach der ID nützlich wäre. Für die einzelnen Felder können verschiedene Filter gewählt werden, bei denen zusätzlich noch das Verhalten über das Attribut *lookup_expr* definiert werden kann. Folgende Filter wurden eingesetzt:

- *CharFilter*: Der CharFilter wird für Text verwendet. Als *lookup expression* können

etwa die Werte *icontains* oder *iexact* gewählt werden, je nachdem ob der Suchbegriff enthalten sein soll, oder ob nur exakte Werte herausgefiltert werden sollen.

- *RangeFilter*: Ein RangeFilter wird für Zahlenwerte verwendet und bietet die Möglichkeit, Minimum und Maximum anzugeben um einen Wert dazwischen herauszufiltern. Dieser Filter wurde für Zahlenwerte verwendet. Wenn der Wert ohne Minimum und Maximum in der Datenbank gespeichert wird, wie beispielsweise der Antennendurchmesser, konnte die Standard-Filtermethode des RangeFilter verwendet werden. Für Werte, die mit Minimum und Maximum gespeichert werden, mussten separate Filtermethoden implementiert werden, da die Werte zweier Attribute zur Filterung herangezogen werden müssen.

- *BooleanFilter*: Dieser Filter ermöglicht es, boolean-Werte zu filtern. Er wurde lediglich verwendet um nach mobilen Antennen zu filtern.

- *ChoiceFilter*: ChoiceFilter werden verwendet, wenn nach einem bestimmten Wert innerhalb einer Menge an vorgegebenen Werten gesucht werden soll. Diese Filterart wurde etwa für die Combiner der Antennenbänder verwendet, da dort nur die Möglichkeiten *Yes* und *No* bestehen.

- *MultipleChoiceFilter*: Dieser Filter erweitert die ChoiceFilter um die Möglichkeit, nach mehreren Werten zu Filtern. MultipleChoiceFilter wurden für die Servicenamen eingesetzt, da eine Antenne über mehrere Services verfügen kann. Um die Antwortmöglichkeiten zu ermitteln, musste die ___*init*___-Methode des Filterset überschrieben werden, um zur Laufzeit zu prüfen, welche Servicenamen in der Datenbank vorkommen. Dadurch können die Auswahlmöglichkeiten des Filters dynamisch auf Grundlage der vorhandenen Daten festgelegt werden. Um nach Antennenbändern zu filtern wurde nach einer ersten Feedback-Runde ebenfalls eine Mehrfachauswahl vorgegebener Bandtypen implementiert. Allerdings sind die Auswahlmöglichkeiten hier statisch.

Eine zusätzliche Anforderung an die Anwendung bestand darin, bestimmte Werte als änderbar zu kennzeichnen, wenn die Möglichkeit besteht, bestimmte Werte für eine zukünftige Satellitenmission gezielt anzupassen. Für diesen Zweck wurden im Datenmodell für bestimmte Werte boolean-Variablen hinzugefügt, um festzulegen, ob eine Änderung möglich ist. Diese Werte gilt es, bei der Filterung zu berücksichtigen. Daher wurden spezielle Filterfunktionen für die entsprechenden Werte implementiert, die über das Attribut *method* den entsprechenden Filtern zugewiesen wurden.
Die angepassten Filter prüfen, ob der entsprechende Wert zur Sucheingabe passt, oder der Wert als änderbar gekennzeichnet ist. Trifft eine der beiden Bedingungen zu, wird die entsprechende Antenne in die Ergebnisliste übernommen. Damit in der Ergebnistabelle nachvollziehbar ist, warum die Antenne trotz unpassender Werte angezeigt wird, wird zusätzlich angezeigt, dass der Wert änderbar ist.
Nachdem eine erste Version der Software zugänglich gemacht wurde, ergab das Feedback, dass das Verhalten der Filter für Downlink, Uplink, G/T und EIRP in manchen Fällen fehlerhaft war. Da zunächst für jeden dieser Werte ein separater Filter für Minimum und

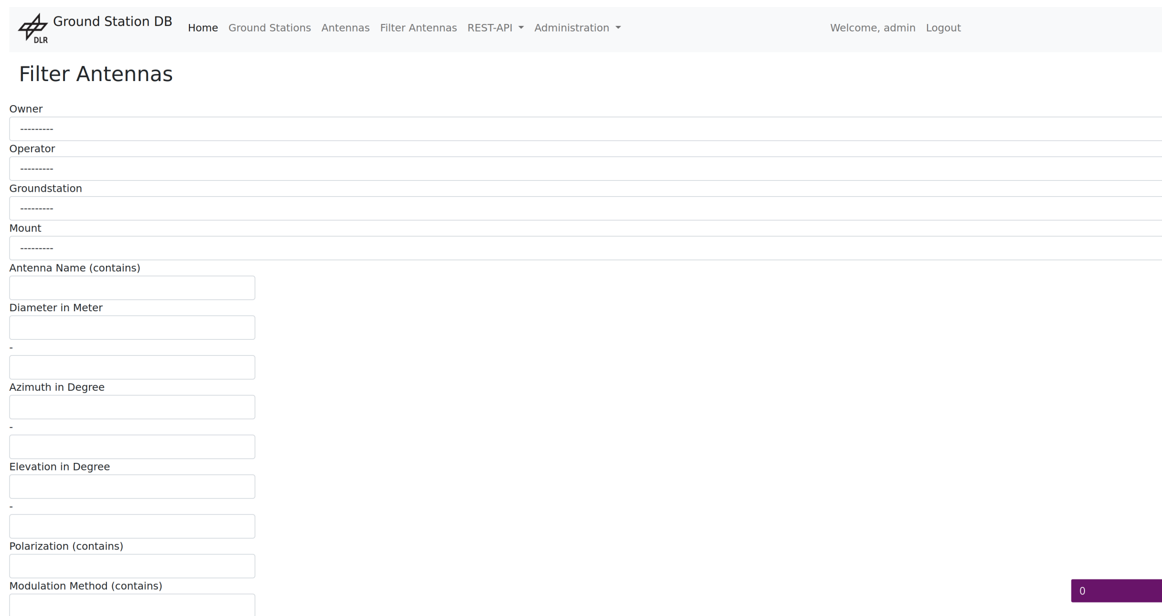


Abbildung 5.4: Eingabeformular zum Filtern der Antennen

Maximum erstellt wurde, kam es bei Eingabe von sowohl Minimum als auch Maximum vor, dass Antennen im Ergebnis auftauchten, bei denen ein Antennenband die Minimum-Bedingung und das andere Antennenband die Maximum-Bedingung erfüllte, während beide Bänder nicht innerhalb der Reichweite von Minimum bis Maximum lagen. Daher wurden neue Filter erstellt, mit denen geprüft werden kann, ob der entsprechende Wert eines Antennenbandes innerhalb der festgelegten Reichweite liegt.

5.7.2 Darstellung in der Webanwendung

Die Funktionalität des Filters der Webanwendung wurde in der Datei *filter.py* im Modul *groundstations* angelegt. Darin wurde die Filterset-Klasse *AntennaFilter* erstellt, die wie im vorherigen Teilabschnitt beschrieben Filter und Filterfunktionen für die einzelnen Attribute festlegt.

Zur Implementierung des Filters innerhalb der Webanwendung wurde eine zusätzliche View-Funktion erstellt. Diese Funktion prüft zunächst die Leseberechtigung des Nutzers. Ist diese vorhanden, wird ein Queryset aus allen Antennen erstellt und dem *AntennaFilter* übergeben. Das Filterobjekt wird anschließend dem Template übergeben. Die Darstellung im Template wird durch *django-filter* erleichtert, da das Filterobjekt ein Eingabeformular beinhaltet, siehe Abbildung 5.4. Unter dem Eingabeformular wurde eine Tabelle angelegt, die anhand des Querysets des Filters dynamisch mit den Filterergebnissen befüllt werden kann. Nach dem Aufruf der Filterseite befinden sich zunächst alle Antennen der Datenbank in der Tabelle. Nach Eingabe beliebiger Filterwerte und einem Klick auf den Submit-Button wird die Tabelle entsprechend angepasst.

5.7.3 Filtern über die REST-Schnittstelle

Die Einbindung des Filters in der Schnittstelle erfolgt über die separate Filterset-Klasse *APIAntennaFilter* in der Datei *filter.py* im Modul *api*. Die Implementierung ist identisch zur Klasse *AntennaFilter*, lediglich die Klasse von der geerbt wird unterscheidet sich.
Um die Klasse in die Schnittstelle einzubinden muss das Filterset lediglich der Klasse *AntennaViewSet* als *filterset_class* übergeben werden. Weiterhin muss in der Konfigurationsdatei *settings.py* unter *REST_FRAMEWORK* das Filterbackend aus dem *django-filters*-Paket als Standardfilterbackend festgelegt werden. Weitere Einstellungen sind nicht nötig.
Das Filtern über die Schnittstelle funktioniert einerseits in der Schnittstellenansicht der Webanwendung unter *host:/api/antennas/*, wo in der oberen rechten Ecke der Button *Filters* angezeigt wird, siehe Abbildung 5.3. Darüber lässt sich ein Filterformular aufrufen, über das wie auf der Webseite gefiltert werden kann. Diese Variante ist jedoch nicht praktikabel für gekoppelte Anwendungen, deshalb ist es zudem möglich über die URL zu filtern. Es folgen Beispieleingaben, um dies zu veranschaulichen:

- *host:/api/antennas/?diameter_in_m_max=20*: Zeigt alle Antennen mit einem maximalen Durchmesser von 20 m an.
- *host:/api/antennas/?diameter_in_m_min=15&diameter_in_m_max=20*: Zeigt alle Antennen mit einem Durchmesser zwischen 15 m und 20 m an.

Auf diese Art lassen sich beliebig lange Kombinationen von Werten erstellen, anhand derer gefiltert werden kann. Bei verschachtelten Werten, die Antennenbändern oder Services zugeordnet sind, muss die Eingabe etwas angepasst werden:

- *host:/api/antennas/?antennaband___band_type=S-Band*: Zeigt alle Antennen an, die über ein S-Band verfügen.
- *host:/api/antennas/?service___service_name=Doppler*: Zeigt alle Antennen an, die Doppler zur Verfügung stellen.

5.7.4 Filtern der Logs

Nachdem die Anwendung zum Testen freigegeben wurde, ergab das Feedback, dass auch für das Änderungslog eine Filtermöglichkeit wünschenswert wäre. Daher wurde innerhalb des Moduls *webapp* eine weitere Datei *filter.py* angelegt. Innerhalb der Datei wurde die Klasse *LogFilter* erstellt. Dabei wurde wie bei der Erstellung der Antennen-Filter vorgegangen. Folgende Filter wurden erstellt:

- *Date*: Filtern nach folgenden festgesetzten Zeiträumen: Heute, Gestern, letzte Woche, letzter Monat, letztes Jahr.
- *User*: Filtert nach dem Nutzer, der die Änderung vollzogen hat. Der Filter sucht mittels *contains*, also muss nicht der exakte Name des Nutzers eingegeben werden.

Abbildung 5.5: Änderungslog mit Eingabeformular

- *Object*: Filtert nach dem Namen des Objekts. Die Eingabe muss nicht exakt erfolgen.
- *Model*: Filtert nach Modellklassen. Es wurden Auswahlmöglichkeiten für die sieben in der Anwendung genutzten Modellklassen vorgegeben.
- *Action*: Filtert nach den Änderungsaktionen CREATE, UPDATE und DELETE.

Das Filterformular wurde über der Ergebnistabelle eingebunden, siehe Abbildung 5.5.

5.8 Erscheinungsbild

5.8.1 Allgemeines

Um das Erscheinungsbild der Anwendung zu verbessern, wurde auf externe Lösungen zurückgegriffen. Für die Darstellung der einzelnen HTML-Elemente wurde das Framework Bootstrap verwendet [4]. Bootstrap ist unter der MIT-Lizenz frei verfügbar. Zur Einbindung mussten lediglich zwei Links im HTML-Head angegeben werden. Dies wurde in der Datei *base.html* durchgeführt, da alle HTML-Templates dieses Projektes als Erweiterungen der Base-Datei angelegt wurden und somit die Ressourcen nutzen können, die dort im Head verlinkt sind. Anschließend konnte in den einzelnen HTML-Tags auf Bootstrapklassen zurückgegriffen werden, um beispielsweise das Styling der Tabellen und Schaltflächen zu optimieren.
Um zusätzliche allgemeine Stylinganpassungen vorzunehmen wurde die Datei *base.css* erstellt. Weiterhin wurde ein Bootstrap Template für die Navigationsleiste am oberen

Bildschirmrand übernommen. Für das weitere Styling der Navigationsleiste wurde die Datei *navbar.css* erstellt.

5.8.2 Cesium

Im Rahmen der ersten Vorführung eines frühen Prototyps wurde der Wunsch geäußert, zusätzlich zu der tabellarischen Darstellung der Antennen eine Weltkarte anzuzeigen, in der die einzelnen Antennenstandorte eingezeichnet würden. Dadurch sollte die Auswahl der Antennen erleichtert werden, da die Darstellung der Standorte auf einer Weltkarte nutzerfreundlicher ist, als die bloße Angabe der Werte für Längen- und Breitengrade.
Um diese Weltkarte zu implementieren, wurde auf Cesium zurückgegriffen. Cesium wird für unterschiedliche Anwendungszwecke angeboten. Im Rahmen dieser Anwendung wurde auf CesiumJS zurückgegriffen, das für die Anwendung in Webapplikationen ausgelegt ist [5]. Die Nutzung ist im Rahmen privater Projekte sowie bei Vorhaben, die Forschung und Bildung dienen, kostenfrei. Bereits die in Abschnitt 2.1 erwähnten bestehenden Projekte griffen zur Visualisierung auf Cesium zurück. Allerdings wurden dort komplexere Ziele verfolgt. Für diese Anwendung war es lediglich erforderlich, eine 2D-Karte anzuzeigen, auf der an den entsprechenden Standorten die Namen der jeweiligen Antennen angezeigt werden, siehe Abbildung 5.6.
Zur Integrierung von Cesium mussten zwei Links im HTML-Head des Dokuments gesetzt werden. Anschließend wurden die Templates *antenna_list.html* und *filter_antennas.html* angepasst. Es wurde ein Cesium Container eingefügt, in dem mithilfe von JavaScript-Code ein Skript ausgeführt wird. Dieser Container enthält ein *AccessToken*, das generiert wurde, nachdem ein Account auf der Cesium-Homepage angelegt wurde. Im Skript wird eine 2D-Weltkarte konfiguriert, die die Namen der vorhandenen Antennen anzeigt. Die Karte ist dreh- und zoombar. Bei Bedarf kann zudem auf eine 3D-Ansicht des Planeten umgeschaltet werden. Als Kartenmaterial können verschiedene Quellen genutzt werden, für diese Anwendung wurde auf das Material von OpenStreetMap zurückgegriffen, da es unter einer freien Lizenz verfügbar ist [25]. In der Filter-Ansicht werden lediglich die aktuell gefilterten Antennen angezeigt, sodass die Namensanzeige auf der Weltkarte nach jeder Filterung aktualisiert wird.

5.9 Unittests

Ein wichtiger Bestandteil der Implementierung ist die Erstellung von Unittests. Der Zweck von Unittests ist es, einzelne Programmbestandteile zu Testen um deren Funktionsfähigkeit zu überprüfen und bei künftigen Programmänderungen sicherzustellen, dass die Funktionsweise erhalten bleibt. Beim Schreiben der Testfälle wurde darauf geachtet, dass die einzelnen Testmethoden möglichst klar, einfach und lesbar sind, da dies wichtige Prinzipien für die Wartbarkeit von Unittests sind [22, S. 124]. Allerdings werden Unittests nicht für alle Programmbestandteile geschrieben. In diesem Projekt wurde beispielsweise darauf verzichtet, das korrekte Anlegen der Modellklassen zu testen, da dies eine Django-Grundfunktionalität darstellt, an der nichts verändert wurde. Auch das Funktionieren einzelner Bibliotheken wird nicht getestet. Dagegen wurde getestet, was

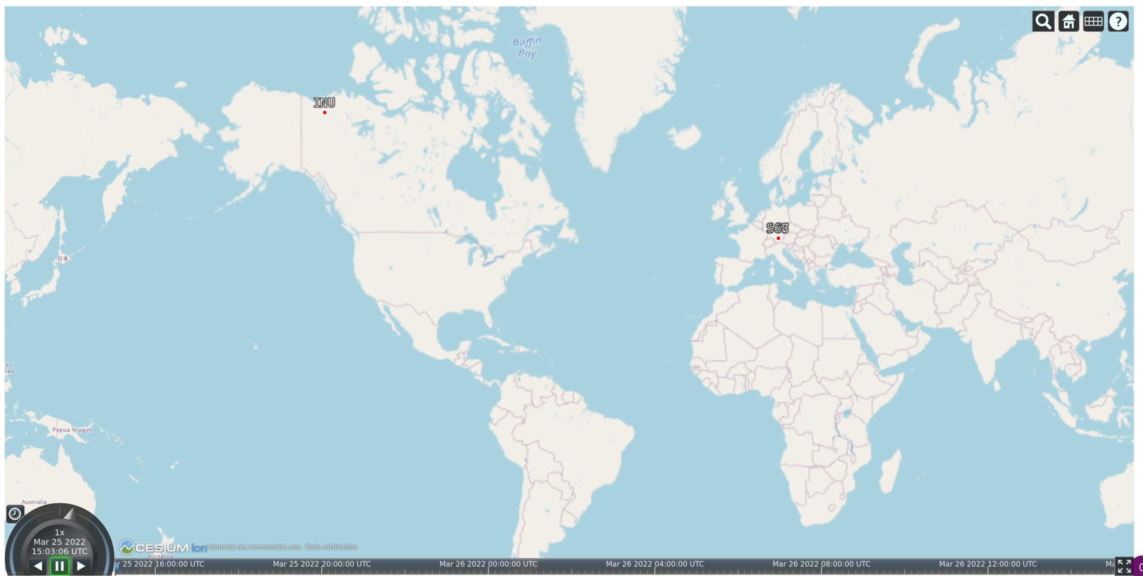


Abbildung 5.6: Cesium-Weltkarte mit eingezeichneten Antennen-Standorten

an zusätzlichen Funktionen im Rahmen dies Projekts entwickelt wurde. Dazu gehören die Views, Eingabeformulare, Filter und auch die Schnittstelle. Die Unittests wurden in den jeweiligen Modulen erstellt. Standardmäßig existiert eine leere Datei *tests.py*. Im Modul *api* wurden die Tests dort erstellt, während in den Modulen *groundstations* und *webapp* mehrere Testdateien erstellt wurden, um die einzelnen Testfälle voneinander abzugrenzen und eine größere Übersichtlichkeit zu gewährleisten.
Beim Testen der Views wurden Testfunktionen erstellt, die anhand des Codes der HTTP-Response prüfen, ob bei entsprechenden Nutzerberechtigungen die Seite geladen wird oder ob der Zugriff verweigert wurde. Weiterhin wurde geprüft, dass das richtige Template für die View-Funktion genutzt wurde.
Die Eingabeformulare wurden getestet, indem verschiedene Kombinationen von validen und invaliden Eingabedaten übergeben wurden, um zu prüfen, ob die Formulare korrekt validiert werden. Damit soll sichergestellt werden, dass keine fehlerhaften Daten in die Datenbank gelangen.
Zum Testen der Filterklassen wurden Beispieldaten erstellt. Anschließend wurden dem jeweiligen Filter verschiedene Kombinationen an Parametern übergeben, um zu prüfen, ob die korrekte Menge an Ergebnissen herausgefiltert wird.
Die Schnittstelle wurde getestet, indem in jeder Testfunktion ein Token abgefragt wurde, und der anschließende Test mit diesem Token durchgeführt wurde. Die Tests umfassten verschiedene Operationen an den Datenbeständen wie hinzufügen und löschen von Objekten oder Listen von Objekten. Weiterhin wurde getestet, ob die Authentifizierung richtig funktioniert, d.h. dass nur Nutzer mit entsprechenden Rechten Schreiboperationen über die API vornehmen konnten. Zudem wurden verschiedene Filterparameter übergeben, um das Funktionieren der Filterfunktion zu gewährleisten.

Kapitel 6

Evaluierung

6.1 Nutzerbefragung

Um die Tauglichkeit der Anwendung hinsichtlich der aufgestellten Anforderungen zu beurteilen, ist es notwendig, die Anwendung durch die zukünftigen Nutzer beurteilen zu lassen. Um dies durchzuführen, wurde eine Testversion zum freien Testen zur Verfügung gestellt.
Die Auslieferung der Testversion erfolgte im DLR-Intranet. Auf einem Server wurde eine virtuelle Maschine erstellt. Zur Auslieferung der Anwendung wurden Docker und Docker-Compose eingesetzt [10]. Docker ermöglicht die schnelle und einfache Auslieferung von Anwendungen. Dazu wird in sogenannten Containern lediglich die für die Ausführung der Anwendung benötigte Software gebündelt. Dadurch ist Docker leichtgewichtiger als eine herkömmliche virtuelle Maschine. Zur Erstellung eines Docker Containers wird lediglich ein sogenanntes *Dockerfile* benötigt, über das der Container konfiguriert wird. Auf dem Rechner, auf dem der Container ausgeführt werden soll, muss lediglich die Docker-Engine installiert sein.
Für die Auslieferung dieses Projektes werden zwei Container benötigt, da die Anwendung und die Datenbank in separaten Containern laufen müssen. Zur Orchestrierung mehrerer Container wird Docker Compose verwendet. Deshalb wird neben der Dockerfile zusätzlich eine Datei namens *docker-compose.yml* benötigt, in der die nötigen Konfigurationen vorgenommen werden. Beide Dateien können im Wurzelverzeichnis des Projektes eingesehen werden.
Zum Testen der Anwendung wurden Nutzeraccounts angelegt und mit verschiedenen Berechtigungen versehen. Diese wurden einigen Mitarbeitern von GSOC und CEF zur Verfügung gestellt, sodass diese Testen konnten, inwiefern die Anwendung den Ansprüchen genügt.
Die Nutzer wurden gebeten, folgende Fragen zu beantworten:

1. Wurden folgende Anforderungen geeignet umgesetzt?
    (a) Anzeige der Daten in Tabellen
    (b) Eingabe/Änderung von Daten
    (c) Filtern in der Anwendung

   (d) Import/Export/Filtern über die REST-Schnittstelle
   (e) Logging der Änderungen
   (f) Nutzerverwaltung/Berechtigungen

2. Darstellung
   (a) Werden die Tabellen/Eingabeformulare übersichtlich dargestellt?
   (b) Was ist positiv, was negativ?

3. Sonstiges: Was ist positiv, was negativ?

6.2 Auswertung

Die Testaccounts wurden mehreren Mitarbeitern von CEF und GSOC zur Verfügung gestellt. Allerdings konnte nur ein Mitarbeiter des GSOC die Zeit zum Testen und für ein Feedbackgespräch freimachen. Dieser Mitarbeiter war jedoch eng in die Anforderungserstellung eingebunden, insofern konnte er wertvolles Feedback liefern.
Nach einigem Testen der Anwendung wurde bestätigt, dass die einzelnen Anwendungsfälle geeignet umgesetzt wurden. Die Anzeige der Daten in Tabellen und die Dateneingabe über Formulare wurde als zweckdienlich beurteilt. Die REST-Schnittstelle wurde ebenfalls getestet und positiv bewertet, da die benötigten Features vorhanden waren. Die Nutzerverwaltung wurde nicht eingehend beurteilt, da der Fokus des Mitarbeiters auf den Programmfunktionalitäten lag, die für seine Arbeit relevant wären.
Die Darstellung wurde positiv bewertet. Die Tabellen seien übersichtlich und die Eingabeformulare nutzerfreundlich. Auch der Aufbau der Seiten sei intuitiv, sodass wenig Fragen bei der Bedienung blieben. Ein Diskussionspunkt war die Tabelle in der Antennenübersicht. Hier wurde die Frage aufgeworfen, ob nicht mehr Daten hätten angezeigt werden können, ähnlich der Ergebnistabelle der Filterseite. Allerdings bestand anschließend ein Konsens darüber, dass die Übersichtlichkeit bei Dutzenden Antennen schnell leiden würde. Es müssten Filter eingefügt werden, um die Übersichtlichkeit zu verbessern, was zu Redundanz führen würde. Da diese Funktionalität bereits in der Filteransicht umgesetzt wurde, sei die aktuelle Darstellung schlussendlich in Ordnung.
Es wurden zudem Hinweise gegeben. So wurde ein fehlerhaftes Verhalten einiger Filter bemerkt, das durch dieses Feedback behoben werden konnte. Weiterhin wurde angemerkt, dass eine Filterfunktion für die Tabelle der Änderungslogs hilfreich wäre, worauf diese ebenfalls implementiert wurde.
Das Testen der Anwendung führte im abschließenden Feedbackgespräch dazu, dass eine umgesetzte Anforderung nachträglich gestrichen wurde. Dabei handelt es sich um die Möglichkeit, bestimmte Werte als änderbar zu kennzeichnen. Der Grundgedanke war, dass auch Antennen herausgefiltert werden sollten, die zwar nicht die gesuchten Werte haben, aber noch angepasst werden könnten um die Anforderungen zu erfüllen. In der Realität wären aber bestimmte Werte wie Codingmethode oder Modulationsmethode immer auf „true“ gesetzt gewesen, da hierbei zur Änderung teilweise nur der Kauf einer

bestimmten Lizenz nötig wäre. Somit wäre ein Filtern nach bestimmten Codingmethoden überflüssig geworden, da sowieso alle Antennen ausgewählt worden wären. Bei anderen Werten wie dem EIRP oder dem G/T hätte sich die Änderbarkeit auf eine Anpassung bzw. Feinjustierung innerhalb eines bestimmten Rahmens bezogen, weshalb auch hier Werte gefiltert worden wären, die nicht gepasst hätten. Die Umsetzung der Anforderung war wie gewünscht erfolgt, allerdings hat der Test ergeben, dass kein Mehrwert für die Nutzung entstand, sondern im Gegenteil viele unpassende Antennen herausgefiltert wurden. Daher wurde beschlossen, die Anforderung nachträglich zu streichen und die Anwendung entsprechend anzupassen. Die entsprechenden boolean-Werte wurden aus den Datenmodellen gestrichen und die betroffenen Tabellen und Filter angepasst.

6.3 Testabdeckung

Weiterer Bestandteil der Evaluierung der Anwendung war die in Abschnitt 5.9 beschriebene Implementierung von Unittests. Um die Testabdeckung zu ermitteln, wurde zusätzlich das Paket *coverage.py* installiert. *Coverage.py* ist ein Tool um die Codeabdeckung in Python-Programmen zu bestimmen [3]. Dadurch kann die Effektivität der Tests bestimmt werden, da zudem angezeigt werden kann, welche Teile des Codes durch Tests ausgeführt werden und welche nicht.

Für die Anwendung wurden 150 Testfunktionen erstellt und eine Testabdeckung in Höhe von 90% ermittelt. *Coverage.py* liefert neben dieser Gesamtabdeckung die einzelnen Quoten zu verschiedenen Dateien. Beispielsweise lag die Testabdeckung für die Datei *groundstations/views.py* bei 53%, während die Abdeckung von *groundstations/filter.py* bei 100% lag. Auffällig war, dass die Abdeckungsquote mit 90% insgesamt sehr hoch lag. Um diesen Wert zu validieren wurde ein weiteres Paket namens *django-nose* über den Python Package Installer installiert [2]. Dieses Paket liefert einen separaten Test Runner sowie die Möglichkeiten, gezielt einzelne Module auszuwählen, um die Abdeckung zu berechnen.

Zur Ermittlung der Abdeckung mit *django-nose* wurden lediglich die drei Django-Apps *api*, *groundstations* und *webapp* ausgewählt. Dadurch, dass der Gesamtumfang verändert wurde, änderte sich die Gesamtquote, welche nun mit einer Abdeckung von 79% angezeigt wurde. Die Werte der einzelnen Dateien blieben jedoch unverändert, weshalb diese als valide angesehen werden können. Die geringste Abdeckung wiesen die Dateien *groundstations/views.py* mit 53% und *webapp/views.py* mit 57% auf. Der Grund dafür ist wahrscheinlich, dass nur getestet wird ob die Views per GET-Request aufgerufen werden können und das richtige Template verwenden. Code, der mit POST-Requests zusammenhängt, also mit dem Validieren und Speichern von Eingabeformularen, wurde nicht getestet, da das richtige Funktionieren der Eingabeformulare separat getestet wurde. Der Großteil der Dateien wies jedoch Werte von 80 bis 100% auf, weshalb die Testabdeckung insgesamt als ausreichend beurteilt werden kann.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, den Austausch zwischen verschiedenen Standorten des DLR vor allem in den frühen Phasen der Planung eines Raumfahrzeugs zu verbessern. Bisher wurden Daten zu Antennen und Bodenstationen vom GSOC in einzelnen Dateien gesammelt und verwaltet. Zu bestimmten Planungszeitpunkten ist es erforderlich, diese Daten mit der CEF zu teilen, wobei die Kontaktaufnahme hauptsächlich telefonisch erfolgte. Zur Verbesserung des Datenaustauschs sollte daher eine Software entwickelt werden, in der ein Standort die Pflege der Daten übernimmt, während dem anderen Standort der Zugriff darauf ermöglicht wird.

Im Rahmen dieser Bachelorarbeit wurden die Anforderungen für diese Software ermittelt. Die Software wurde modelliert und anschließend implementiert. Auf die Implementierung folgte eine Testphase zur Evaluierung der Funktionalität der Anwendung. Diese Testphase führte zu weiteren Anpassungen sowohl der Anwendung als auch der Anforderungen an diese. Das Feedback hat ergeben, dass die Anwendung die geforderten Anwendungsfälle umsetzt.

Zukünftig könnte die Anwendung um weitere Funktionalitäten erweitert werden, was durch den modularen Aufbau des Django-Frameworks erleichtert wird. Beispielsweise könnten Funktionalitäten aus der in Abschnitt 2.1 erwähnten Anwendung Orbitscalc übernommen werden, sodass auch Bahnparameter eines Satelliten eingegeben werden könnten um die Datenübertragung zwischen Satellit und Antenne zu berechnen. Weiterhin könnten andere Datenbanken angeschlossen werden. Die SANA (Space Assigned Numbers Authority) führt beispielsweise verschiedene Registries, deren Einträge ebenfalls über eine REST-Schnittstelle abgerufen werden können [30]. Auch ein Datenaustausch mit Raumfahrtagenturen kooperierender Länder könnte sich anbieten.

Abseits inhaltlicher Erweiterungen bietet die Anwendung noch Raum für softwaretechnische Verbesserungen. Beispielsweise ist es einem Nutzer aktuell nicht möglich, sein Passwort selbst zurückzusetzen. Momentan muss dazu ein Admin kontaktiert werden, der das Nutzerpasswort in der Admin-Ansicht händisch zurücksetzt. Ein weiteres Beispiel für Verbesserungsmöglichkeiten betrifft die Token-Authentifizierung über die REST-Schnittstelle. In der derzeitigen Fassung behalten diese Tokens dauerhaft Gültigkeit. Um die Sicherheit zu erhöhen wäre es jedoch geboten, einen Gültigkeitszeitraum für die Tokens festzulegen, sodass diese regelmäßig erneuert werden müssen. Zudem wäre es

unter Umständen hilfreich, wenn die Datenbank über eine *Undo*- oder *Reset*-Funktion verfügen würde, mithilfe derer ein bestimmter älterer Zustand bequem wiederhergestellt werden könnte.

Abkürzungsverzeichnis

API Application Programming Interface

CEF Concurrent Engineering Facility

DLR Deutsches Zentrum für Luft- und Raumfahrt

EIRP Equivalent Isotropic Radiated Power

GSOC German Space Operations Center

G/T Gain-to-noise Temperature

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

MTV Model Template View

MVC Model View Controller

REST Representational State Transfer

SANA Space Assigned Numbers Authority

SOAP Simple Object Access Protocol

SQL Structured Query Language

UML Unified Modeling Language

URL Uniform Resource Locator

XML Extensible Markup Language

Literaturverzeichnis

[1] Martin Alderete. Lösungsvorschlag zu django-auditlog issue 115. Abgerufen am 04. März 2022. URL: https://github.com/jazzband/django-auditlog/issues/115#issuecomment-572278206.

[2] Jeff Balogh, Erik Rose, James Socol, Rob Hudson und John Whitlock. django-nose. Abgerufen am 28. März 2022. URL: https://github.com/jazzband/django-nose.

[3] Ned Batchelder. Coverage.py. Abgerufen am 28. März 2022. URL: https://coverage.readthedocs.io/en/6.3.2/index.html.

[4] Bootstrap. Bootstrap. Abgerufen am 10. März 2022. URL: https://getbootstrap.com/.

[5] Cesium GS Inc. Cesiumjs. Abgerufen am 10. März 2022. URL: https://cesium.com/platform/cesiumjs/.

[6] Hannes Diener. Orbitscalc. Abgerufen am 21. Oktober 2021. URL: https://github.com/Surf1sh/Orbitscalc.

[7] Django Software Foundation. Django documentation - model field reference. Abgerufen am 09. Dezember 2021. URL: https://docs.djangoproject.com/en/3.2/ref/models/fields/.

[8] Django Software Foundation. Django documentation - writing and running tests. Abgerufen am 10. Januar 2022. URL: https://docs.djangoproject.com/en/4.0/topics/testing/overview/.

[9] Django Software Foundation. Django project. Abgerufen am 28. November 2021. URL: https://www.djangoproject.com/.

[10] Docker Inc. Docker. Abgerufen am 06. März 2022. URL: https://www.docker.com/.

[11] Navnath Gadakh. Why django is so impressive for developing with postgresql and python. Abgerufen am 29. November 2021. URL: https://www.enterprisedb.com/postgres-tutorials/why-django-so-impressive-developing-postgresql-and-python.

[12] Alex Gaynor und Carlton Gibson. django-filter. Abgerufen am 05. März 2022. URL: https://django-filter.readthedocs.io/en/stable/index.html.

[13] Nigel George. Django's structure—a heretic's eye view. Abgerufen am 10. Januar 2022. URL: https://djangobook.com/mdj2-django-structure/.

[14] Marcin Gnat und Michael Schmidhuber. Communication and Infrastructure. In Thomas Uhlig, Florian Sellmaier und Michael Schmidhuber, editors, *Spacecraft Operations*, pages 91–118. Springer Vienna, 2015. doi:10.1007/978-3-7091-1803-0_3.

[15] Rinu Gour. Working structure of django mtv architecture. Abgerufen am 11. Januar 2022. URL: https://towardsdatascience.com/working-structure-of-django-mtv-architecture-a741c8c64082.

[16] Tom Gross und Michael Koch. *Computer-Supported Cooperative Work*. Oldenbourg Wissenschaftsverlag, 2009.

[17] Adam Johnson. How to unit test a django form. Abgerufen am 17. Januar 2022. URL: https://adamj.eu/tech/2020/06/15/how-to-unit-test-a-django-form/.

[18] Alain Karas, Pascal Cousin, Gerard Kipfer und Arnaud Robert. *A triband autotrack concentric feed S, X, Ka for ground stations related to earth observation*. 2018. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2018-2555, doi:10.2514/6.2018-2555.

[19] Jan-Kelle Kester. django-auditlog documentation. Abgerufen am 04. März 2022. URL: https://django-auditlog.readthedocs.io/en/latest/index.html.

[20] Wilfried Ley, Klaus Wittmann und Willi Hallmann. *Handbook of Space Technology*. 2009. doi:10.1002/9780470742433.

[21] Abdul Majeed und Ibtisam Rauf. MVC Architecture: A Detailed Insight to the Modern Web Applications Development. 2018.

[22] Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftmanship*. Pearson Education, 2009.

[23] Object Management Group. Unified modelling language. Abgerufen am 29. November 2021. URL: https://www.uml.org/.

[24] OpenJS Foundation. Node.js. Abgerufen am 27. Januar 2022. URL: https://nodejs.org/en/.

[25] OpenStreetMap Foundation. Openstreetmap. Abgerufen am 26. März 2022. URL: https://www.openstreetmap.org/#map=6/51.330/10.453.

[26] Taylor Otwell. Laravel. Abgerufen am 27. Januar 2022. URL: https://laravel.com/.

[27] Postman. Postman. Abgerufen am 08. Februar 2022. URL: https://www.postman.com/.

[28] Python Software Foundation. unittest - unit testing framework. Abgerufen am 29. November 2021. URL: https://docs.python.org/3/library/unittest.html#module-unittest.

[29] Tobias Schlauch, Michael Meinel und Carina Haupt. Dlr software engineering guidelines, August 2018.

[30] Space Assigned Numbers Authority. Service sites and apertures. Abgerufen am 24. März 2022. URL: https://sanaregistry.org/r/service_sites_apertures/.

[31] Andrew S. Tanenbaum und Maarten Van Steen. *Distributed Systems - Principles and Paradigms*. Pearson Education, 2007.

[32] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. Abgerufen am 29. November 2021. URL: https://www.postgresql.org/.

[33] Juris Tihomirovs und Jānis Grabis. Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics. *Information Technology and Management Science*, 19(1), 2017. doi:10.1515/itms-2016-0017.

[34] Thomas Uhlig, Florian Sellmaier und Michael Schmidhuber. *Spacecraft operations*. 2015. doi:10.1007/978-3-7091-1803-0.

[35] Wikipedia. Model view controller. Abgerufen am 09. Februar 2022. URL: https://de.wikipedia.org/wiki/Model_View_Controller.