Autor: Matthias Regner **Datum:** 19.05.2025

Softwaredokumentation des PIO-ULB Editors nach dem Arc42 Standard

Forschungsprojekt CARE REGIO – Teilprojekt 3: DigiPÜB

PIO-ULB Spezifikationsversion:	1.0.0
PIO-ULB Editor Version:	1.2.2
Projektleitung:	Prof. Dr.-Ing. Alexandra Teynor, Prof. Dr. Claudia Reuter
Projektmitarbeiter:	Matthias Regner, Lukas Kleybolte, Viktor Werlitz, Elisabeth Veronika Mess, Sabahudin Balic, Adrian Petters

CARE REGIO gemeinsam. innovativ. digital.

Technische Hochschule Augsburg

UNIVERSITÄTSKLINIKUM AUGSBURG

Inhaltsverzeichnis

1. Einführung und Ziele 6

2. Randbedingungen 9

3. User Interface des PIO-ULB Editors 12

4. Kontextabgrenzung 16

5. Lösungsstrategien 18

6. Bausteinsicht

6.1 Subsystem: Meta-Repository 20

6.2 Subsystem: FHIR-Repository 21

6.3 Subsystem: Backend-Server 25

6.4 Subsystem: Frontend-Server 31

6.5 Subsystem: Validator Service 36

7. Laufzeitsicht

7.1 Login eines Nutzers 38

7.2 Logout eines Nutzers 39

7.3 Neuen PIO-ULB anlegen 39

7.4 Dateneingabe in einen PIO-ULB 41

7.5 Anlegen eines neuen Adressbuch-Eintrags 41

7.6 Export eines PIO-ULBs im XML- bzw. PDF-Format / Serialisierungslogik 43

7.7 Import eines PIO-ULBs im XML-Format / Deserialisierungslogik 46

8. Verteilungssicht 51

9. Querschnittliche Konzepte

9.1 PrimitiveDataType Klassen 53

9.2 SubTree Klasse 54

9.3 Struktur des Redux Stores 55

9.4 Formulare zur Dateneingabe im Frontend 57

10. Architekturentscheidungen 60

11. Technische Schulden 63

12. Glossar 66

Abbildungsverzeichnis

Abbildung 1: Schematische Darstellung eines FHIR-Bundles 9

Abbildung 2: Darstellung der Patientenadresse im XML-Format 10

Abbildung 3: Anmelde-Screen des PIO-ULB Editors 12

Abbildung 4: Home-Screen des PIO-ULB Editors 13

Abbildung 5: Editoransicht des PIO-ULB Editors mit einklappbarem Seitenmenü 13

Abbildung 6: Auswahlmöglichkeiten der Exportfunktion 14

Abbildung 7: Adressbuch-Screen des PIO-ULB Editors 14

Abbildung 8: Dritter Tab zur Anzeige von Daten, die nicht im PIO-Small enthalten sind 15

Abbildung 9: Nutzerdaten im Kopfzeilenmenü inkl. Logout Button 15

Abbildung 10: Kontext-Diagramm des PIO-ULB Editors 16

Abbildung 11: UML-Komponentendiagramm des PIO-ULB Editors 18

Abbildung 12: Bestandteile des „Meta-Repositories“ und deren Zusammenspiel 20

Abbildung 13: Bestandteile des „FHIR-Repositories“ und deren Zusammenspiel 22

Abbildung 14: Auszug aus der ResourceLookUpTable.json 23

Abbildung 15: Auszug aus der ValueSetLookUpTable.json 23

Abbildung 16: Auszug aus der TranslationListOfExcludedPaths.json 24

Abbildung 17: Bestandteile des Subsystems „Backend“ und deren Zusammenspiel 25

Abbildung 18: Gekürztes UML-Klassendiagramm vom "Root Object" 27

Abbildung 19: Schematische Darstellung der Patientenadresse als "EntryType" 28

Abbildung 20: Bestandteile des Subsystems „Frontend“ und deren Zusammenspiel 31

Abbildung 21: Hierarchische Struktur der React Komponenten im Frontend inkl. Verweise auf spezielle Dateien 33

Abbildung 22: Bestandteile des „Validator Services“ und deren Zusammenspiel 36

Abbildung 23: Szenario „Login eines Nutzers“ als UML-Sequenzdiagramm 38

Abbildung 24: Szenario „Logout eines Nutzers“ als UML-Sequenzdiagramm 39

Abbildung 25: Szenario „Öffnen eines leeren PIO-ULBs“ als UML-Sequenzdiagramm 40

Abbildung 26: Szenario „Dateneingabe in einen PIO-ULB“ als UML-Sequenzdiagramm 41

Abbildung 27: Szenario „Anlegen eines neuen Adressbuch-Eintrags“ als UML-Sequenzdiagramm 42

Abbildung 28: Szenario „Export einer XML- bzw. PDF-Datei“ als UML-Sequenzdiagramm 43

Abbildung 29: Exportprozess im Backend als UML-Aktivitätsdiagramm 45

Abbildung 30: Szenario „Import einer XML-Datei“ als UML-Sequenzdiagramm 47

Abbildung 31: Importprozess eines XML-Strings im Backend als UML-Aktivitätsdiagramm 48

Abbildung 32: Szenario „fehlerhafter Import einer XML-Datei“ (UML-Sequenzdiagramm) 50

Abbildung 33: UML-Verteilungsdiagramm des PIO-ULB Editors im lokalen Netzwerk einer Einrichtung 51

Abbildung 34: UML-Klassendiagramm für zwei beispielhafte PrimitiveDataTypes 54

Abbildung 35: UML-Klassendiagramm der SubTree Klasse 54

Abbildung 36: Struktur des Redux Stores in der „localVersion“ mit beispielhaftem Inhalt 56

Abbildung 37: Das Formular „Barthelgesamtindex“ im Tab „Pflegerische Infobausteine“ des PIO-ULB Editors 58

Abbildung 38: Das Formular „Risiken“ im Tab „Pflegerische Infobausteine“ des PIO-ULB Editors 58

Abbildung 39: Modal zum Hinzufügen eines neuen Risikos 59

Tabellenverzeichnis

Tabelle 1: Auflistung und Erklärung aller Quellcode-Dateien des Meta-Repositories 21

Tabelle 2: Erläuterungen zu den LookUpTables des FHIR-Repositories 23

Tabelle 3: Auflistung und Erklärung aller Quellcode-Dateien des FHIR-Repositories 24

Tabelle 4: Erläuterung der "localVersion" und "webVersion" 26

Tabelle 5: Auflistung und Erklärung aller Dateien aus dem Quellcode des Backends 30

Tabelle 6: Detaillierte Beschreibung der „Backend Services“ 32

Tabelle 7: Auflistung und Erklärung aller Dateien aus dem Quellcode des Frontends 35

Tabelle 8: Auflistung und Erklärung aller Quellcode-Dateien des Validator Services 37

Tabelle 9: Wichtige Architekturentscheidungen und deren Auswirkungen auf die Softwarearchitektur 60

Abkürzungsverzeichnis

Abkürzung	Bedeutung
API	Application Programming Interface
ePA	Elektronische Patientenakte
FHIR	Fast Healthcare Interoperability Resources
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifikationsnummer
JSON	JavaScript Object Notation
JWT	JSON Web Token
KIM	Kommunikation im Medizinwesen (Service der TI)
KIS	Krankenhausinformationssystem
MIO	Medizinisches Informationsobjekt
NPM	Node Package Manager
PIO	Pflegerisches Informationsobjekt
PIO-ULB	Pflegerisches Informationsobjekt - Pflegeüberleitungsbogen
SCSS	Sassy Cascading Style Sheets
SNOMED	Systematized Nomenclature of Medicine
TI	Telematikinfrastruktur
UI	User Interface (= Benutzeroberfläche)
ULB	Überleitungsbogen
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensible Markup Language

1. Einführung und Ziele

Angesichts des Mangels an qualifiziertem Pflegepersonal in Deutschland und der steigenden Zahl pflegebedürftiger Menschen ist es notwendig, die Pflegeprozesse effizienter zu gestalten. Eine digitale Unterstützung scheint ein vielversprechender Ansatz zu sein, da viele Prozesse im Gesundheitswesen papierbasiert sind, wie zum Beispiel der Pflegeüberleitungsprozess. Der Pflegeüberleitungsprozess ist ein zentraler Prozess im Gesundheitswesen, der eine nahtlose Übergabe von Patienteninformationen zwischen verschiedenen Pflegeeinrichtungen sicherstellt. Hierbei werden relevante Informationen zum Patienten als Überleitungsbericht (ULB) zusammengefasst und von einer Einrichtung in die Nächste übermittelt. Die Berichte beinhalten relevante Angaben zu Diagnosen, Medikation, pflegerische Maßnahmen sowie besondere Hinweise für die Versorgung und mehr.

Meist werden die Patientendaten aus unterschiedlichen Dokumentablagen (Papier und Computer) zusammengetragen und in einen einrichtungsspezifischen, nicht standardisierten ULB übertragen. Die Erstellung des ULB erfolgt also händisch und ist sehr zeitaufwendig. Der ULB wird den Patienten meist erst am Tag der Überleitung als Ausdruck mitgegeben. Manchmal haben die Patienten auch keinen ULB bei sich und die empfangende Einrichtung muss diesen nachfordern. In den meisten Fällen wird der ULB von der sendenden Einrichtung via Fax nachgereicht, was datenschutzrechtlich sehr kritisch ist. Eine frühzeitige Vorbereitung auf den Patienten in der empfangenden Einrichtung ist somit kaum möglich.

Derzeit gibt es in Deutschland weder einen datenschutzkonformen Übertragungsweg für die digitale Übermittlung von ULBs noch einen einheitlichen und etablierten Standard zur Abbildung dieser Berichte. Im letzteren Fall liegt das Problem unter anderem daran, dass die meisten pflegerischen Einrichtungen unterschiedliche Pflegedokumentationssysteme nutzen und bisher keine standardisierte Darstellung etabliert werden konnte. Die daraus resultierenden Inkompatibilitäten schaden der Interoperabilität zwischen Gesundheitseinrichtungen und erfordern eine zeitaufwendige, manuelle Datenübertragung vom ausgedruckten ULB in des digitale Pflegedokumentationssystem der aufnehmenden Einrichtung. Im Kontext des Pflegeüberleitungsprozesses ergeben sich zusammengefasst folgende **Problematiken**:

- Meist zeitaufwendiges Zusammentragen der Patientendaten aus unterschiedlichen Quellen
- Meist manuelles Erstellen des Überleitungsberichts (ULB)
- Papierbasierte Datenübermittlung am Tag der Patientenüberleitung sowie daraus resultierende Unfähigkeit, adäquate Vorbereitungen für den neuen Patienten zu treffen
- Einrichtungsspezifische ULBs stören die Interoperabilität beim Datenaustausch
- Aufwendige, fehleranfällige und manuelle Datenübertragung vom ULB in das Dokumentationssystem der aufnehmenden Einrichtung

All diese Aspekte erschweren ein effizientes Arbeiten der Pflegefachpersonen. Insbesondere das Fehlen eines digitalen Standardformats führt zu einem erhöhten administrativen Mehraufwand. Daher ist es notwendig, den Pflegeüberleitungsprozess durch eine

einheitliche Standardisierung des ULBs und die Digitalisierung der Datenübertragung zu optimieren.

Um die aufgeführten Problemstellungen in Zukunft zu lösen, hat das Bundesministerium für Gesundheit Standardisierungsaufträge für Gesundheitsdokumente vergeben und das deutsche Gesundheitsnetzwerk, die Telematikinfrastruktur (TI), unter anderem auf Pflegeheime ausgeweitet. Im Jahr 2021 wurde im Rahmen des „Digitalen-Versorgung-und-Pflege-Modernisierungs Gesetz“ (DVPMG) der Auftrag erteilt, deutsche Gesundheitsdokumente in Form von medizinischen und pflegerischen Informationsobjekten (**MIOs** & **PIOs**) zu standardisieren und zu digitalisieren. MIOs & PIOs sind strukturell kompatibel zum internationalen FHIR-Standard (Fast Healthcare Interoperability Resources) und werden in XML-Dateien abgelegt. Für ULBs gibt es seit Januar 2023 ein eigenes Format: Den PIO Überleitungsbogen (PIO-ULB), welcher das aller erste PIO für die Pflege ist. Es gibt jedoch keine gesetzliche Frist für die verpflichtende Implementierung dieses Standards. Daher ist ein Großteil der Pflegedokumentationssysteme mit dem neuen PIO-ULB Standard noch nicht kompatibel. Die TI, welche im Jahr 2025 auf alle pflegerischen Einrichtungen erweitert werden soll, ermöglicht das datenschutzkonforme Übermitteln von Patientendaten (z.B. ein ULB) zwischen allen Gesundheitseinrichtungen, die an die TI angebunden sind.

Die bereits unternommenen Bemühungen zur Standardisierung und Digitalisierung des Datenaustauschs zwischen Gesundheitseinrichtungen greift das Teilprojekt DigiPÜB des **Care Regio** Forschungsverbundes auf, um oben beschriebene Problematik im Datenüberleitungsprozess zu adressieren. In diesem Teilprojekt arbeiten die *Technische Hochschule Augsburg*, das *Universitätsklinikum Augsburg* und zwei kooperierenden Pflegeeinrichtungen in Augsburg gemeinsam an der Konzeption und Implementierung eines **PIO-ULB Editors**. Dieser Editor ist eine der ersten frei nutzbaren Implementierungen des PIO-ULB Standards. Der Editor generiert und importiert standardkonforme PIO-ULB Dateien. Ein nutzerfreundliches User Interface erleichtert dem Pflegepersonal das Arbeiten mit dem neuen Standard.

Der PIO-ULB Editor wurde entwickelt, um folgende Ziele zu erreichen:

Mittelfristige Ziele des PIO-ULB Editors:

- Erstellen, Editieren, Exportieren und Importieren von standardkonformen PIO-ULBs
- Gebrauchstaugliche Implementierung des PIO-ULB Standards, um eine beispielhafte, digital gestützte Patientenüberleitung durchzuführen
- Übergangslösung, solange Dokumentationssysteme PIO-ULBs nicht unterstützen
- Diskussionsgrundlage, um Inhalte des neuen PIO-ULB Standards für alle Stakeholder zu visualisieren

Langfristiges Ziel eines digital gestützten Pflegeüberleitungsprozess:

- Automatische Erstellung, Übertragung und Import von Überleitungsberichten im PIO-ULB Format aus und in hauseigene Dokumentationssysteme
- Einsparung von Arbeitszeit des Pflegepersonals

Der PIO-ULB Editor ist als Übergangslösung zu verstehen, da er weiterhin manuelle Arbeitsschritte erfordert, wie zum Beispiel das händische Eintragen der Patientendaten in den Editor bei der sendenden Einrichtung oder das Übertragen der Daten vom Editor in das Dokumentationssystem der aufnehmenden Einrichtung. Allerdings erfolgen diese Arbeitsschritte ausschließlich am Rechner, wodurch kein weiterer Medienbruch entsteht. Erst wenn die Softwarehersteller der Dokumentationssysteme eine Schnittstelle zum Exportieren und Importieren von standardisierten PIO-ULBs implementieren, kann die Datenübertragung vom System der sendenden Einrichtung zum System der aufnehmenden Einrichtung vollkommen automatisch erfolgen.

Darüber hinaus wurde der PIO-ULB Editor nicht für den täglichen Einsatz im pflegerischen Alltag konzipiert. Vielmehr handelt es sich um einen **Prototyp**, der primär zur Erprobung des PIO-ULB Standards dient. Außerdem sollen mit dem Prototyp die oben genannter Ziele adressiert werden. Es ist kein Ziel des Forschungsprojekts, eine marktreife Software zu entwickeln.

2. Randbedingungen

Im Rahmen folgenden Randbedingungen wurde der PIO-ULB Editor entwickelt:

- Konformität zum PIO-ULB Standard
- Konformität zum FHIR Standard
- Umfangsreduktion des PIO-ULB Standards auf das *PIO-Small*
- Keine Schnittstelle zu etablierten Dokumentationssystemen
- Keine Integration in etablierte Dokumentationssysteme
- Primäre Entwicklung für stationäre Pflegeheime und Krankenhäuser
- Konformität zu deutschen Datenschutzbestimmungen

Die wohl wichtigste Randbedingung, unter welcher der PIO-ULB Editor entwickelt wurde, ist die **Konformität zum PIO-ULB Standard**. Der Standard wurde von der Mio42 GmbH (eine 100%ige Tochtergesellschaft der Kassenärztlichen Bundesvereinigung) im gesetzlichen Auftrag spezifiziert. Er schreibt detailliert die Struktur, die Inhalte und die Verwendung von speziellen Codesystemen vor. Das Design der Benutzeroberfläche wurde stark durch die Struktur des Standards geprägt.

Der PIO-ULB Standard basiert auf dem internationalen **FHIR** Standard. Das FHIR-Datenformat ist ein international anerkannter und weit verbreiteter Standard zur maschinenlesbaren Abbildung von medizinischen Daten. FHIR bedient sich einem ressourcenbasierten Konzept, wobei jede Ressource einen Informationsbaustein darstellt (z.B. Patient, Organisation, Arzt, Medikament, Pflegeplan, Allergie, usw.). Die Ressourcen können zu einem Paket (Bundle) inklusive Inhaltsverzeichnis (Composition) zusammengefasst werden, um Patienteninformationen abzubilden (siehe Abb. 1). Dabei bleibt FHIR so flexibel, dass der internationale Standard leicht an spezifische Anwendungskontexte angepasst werden kann.

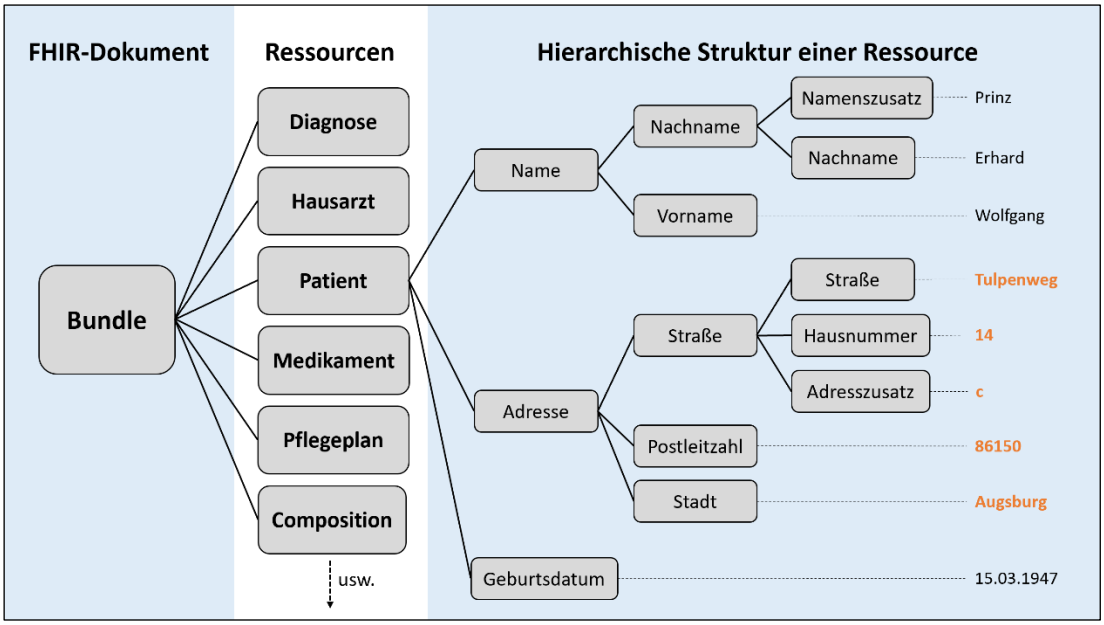

Abbildung 1: *Schematische Darstellung eines FHIR-Bundles (Daten aus Abb. 2 orange markiert)*

Jeder Ressource ist eine statische ID zugeordnet. So haben zum Beispiel zwei Hausarzt Ressourcen jeweils eine eigene ID, damit jede Ressource eindeutig identifizierbar ist. Innerhalb einer Ressource sind die Daten hierarchisch gegliedert und sehr fein granular aufgeteilt. Wie eine spezielle Ressource strukturiert ist, wird von dessen FHIR-Strukturdefinition eindeutig vorgeschrieben. FHIR-Bundles können als JSON- oder XML-Datei abgespeichert werden und sind daher maschinenlesbar. Die Mio42 GmbH legte jedoch bereits fest, dass MIOs & PIOs als XML-Dateien zu implementieren sind.

Abbildung 1 stellt schematisch den Aufbau eines FHIR-Bundles dar. Dabei wird auf die Struktur der Patientenstammdaten (Name, Adresse und Geburtsdatum) genauer eingegangen. Abbildung 2 zeigt die XML-Repräsentation der Adresse, wie sie später im PIO-ULB zu finden ist:

```
<address>
	<type value="postal"/>
	<use value="home"/>
	<text value="Tulpenweg 14 c, 86150 Augsburg, D"/>
	<line value="Tulpenweg 14 c">
		<extension url="http://hl7.org/fhir/StructureDefinition/iso21090-ADXP-streetName">
			<valueString value="Tulpenweg"/>
		</extension>
		<extension url="http://hl7.org/fhir/StructureDefinition/iso21090-ADXP-houseNumber">
			<valueString value="14"/>
		</extension>
		<extension url="http://hl7.org/fhir/StructureDefinition/iso21090-ADXP-additionalLocator">
			<valueString value="c"/>
		</extension>
	</line>
	<postalCode value="86150"/>
	<city value="Augsburg"/>
	<country value="D"/>
</address>
```

Abbildung 2: *Darstellung der Patientenadresse im XML-Format (Daten aus Abb. 1 orange markiert)*

Da der PIO-ULB Standard erst im Januar 2023 final veröffentlicht wurde und das Forschungsprojekt bereits im Jahr 2020 begann, mussten während der Projektlaufzeit mehrere Änderungen am Standard berücksichtigt werden. Verbesserungsvorschläge bezüglich des Standards, welche während des Forschungsprojekts erarbeitet wurden, sind an die Mio42 GmbH weitergeleitet worden. Selbst nach der finalen Veröffentlichung kam es zu geringfügigen Änderungen, weshalb die vom Editor generierte XML-Datei schon heute nicht mehr exakt dem Standard entspricht. Dies betrifft zum Beispiel die Header-Informationen aller FHIR-Ressourcen. Auch in Zukunft wird es wahrscheinlich zu kleinen Änderungen am Standard kommen. Selbst eine groß angelegte Überarbeitung des Standards schließt die Mio42 GmbH nicht aus.

Aufgrund begrenzter personeller Ressourcen im Forschungsprojekt und entsprechender Rückmeldungen des Pflegefachpersonals, wurde eine Aufwand-Nutzen-Abschätzung durchgeführt. Diese ergab, dass der Umfang des PIO-ULB Standards für den pflegerischen Alltag zu umfangreich und komplex ist. Daraufhin wurde beschlossen, dass nur eine sorgfältig ausgewählte Teilmenge des Standards durch den PIO-ULB Editor verarbeitet werden kann. Diese Teilmenge wird im Folgenden als **PIO-Small** bezeichnet. Die Auswahl der Informationen, die letztendlich im *PIO-Small* abgebildet werden, erfolgte anhand fachlicher Expertise der kooperierenden Gesundheitseinrichtungen sowie

technischer Aspekte, um die strukturellen Abhängigkeiten und die daraus resultierende Komplexität des Standards zu reduzieren. Die Inhalte des *PIO-Smalls* sind in einem eigenen Dokument detailliert beschrieben.

Da der PIO-ULB Editor als Übergangslösung konzipiert wurde, bei dem immer noch manuelle Schritte zur Datenverarbeitung notwendig sind, und erst die Kompatibilität von PIO-ULBs mit den Dokumentationssystemen einen gänzlich automatisierten Datenübertragungsprozess ermöglichen, wäre eine Schnittstelle vom Editor zu anderen Dokumentationssystemen sinnvoll, um manuellen Arbeitsaufwand erheblich zu reduzieren. Die Umsetzung einer solchen **Schnittstelle** war aufgrund mehrerer Faktoren nicht möglich. Zum einen wurde die gesetzliche Umsetzungsverpflichtung im zweiten Quartal 2024 zurückgezogen, wodurch sich die Prioritäten der Softwarehersteller verschoben. Zum anderen ist der PIO-ULB Standard noch zu neu, zu unerprobt, und sehr gering bis gar nicht verbreitet, sodass Softwarehersteller mit dessen Implementierung zögern.

Zudem konzentriert sich das Forschungsprojekt auf die Bedürfnisse der kooperierenden Einrichtungen, also stationäre Pflegeheime und Krankenhäuser. In Form von regelmäßigen User-Tests konnte das Pflegepersonal Einfluss auf die iterative Entwicklung des PIO-ULB Editors nehmen. Ambulante Pflegedienste wurden nicht betrachtet.

Eine weitere Rahmenbedingung, in der sich das Forschungsprojekt bewegen musste, sind die **deutschen Datenschutzbestimmungen** für Gesundheitsdaten. Alle Einrichtungen, die mit Gesundheitsdaten von Patienten arbeiten, müssen sicherstellen, dass diese Daten ausschließlich intern verarbeitet und gespeichert werden und die Einrichtung nicht verlassen. Der PIO-ULB Editor konnte also nicht als frei zugängliche Cloudanwendung umgesetzt werden, sondern muss im lokalen Netzwerk der Einrichtungen betrieben werden („*localVersion*“). Trotzdem unterstützt der Editor eine „*webVersion*“, welche für den Betrieb auf einer öffentlichen Webseite zu Demonstrationszwecken optimiert ist. Auf dieser Webseite [1] dürfen zwar keine realen Patientendaten eingetragen werden, jedoch kann das Forschungsprojekt auf diese Weise die Forschungsergebnisse veröffentlichen und für alle Interessenten nutzbar machen. Der Quellcode des gesamten PIO-ULB Editors ist auf GitHub veröffentlicht. [2] Wie der PIO-ULB Editor lokal auf einem Computer installiert werden kann, wird in Kapitel 8 beschrieben.

Als letzte Rahmenbedingung bleibt zu erwähnen, dass der PIO-ULB Editor als eigenständige Software entwickelt wurde. Das langfristige Ziel ist jedoch die Integration des Editors in die Dokumentationssysteme der Einrichtungen. Aus diesem Grund wurde im Editor kein User Management implementiert, weil dies zukünftig vom Dokumentationssystem übernommen werden soll. Der Editor fokussiert sich daher auf zentrale Kernfunktionen, wie das Generieren und Verarbeiten von standardkonformen XML-Dateien.

[1] Technische Hochschule Augsburg, *PIO-ULB Editor*, URL: https://pio-editor.de/ [zuletzt aufgerufen: 10.03.2025].

[2] GitHub, *Technische Hochschule Augsburg – Institut für agile Softwareentwicklung*, URL: https://github.com/THAias [zuletzt aufgerufen: 10.03.2025].

3. User Interface des PIO-ULB Editors

Dieses Kapitel geht auf die wichtigsten Elemente des User Interfaces (UI) ein, um das Verständnis für den PIO-ULB Editor in allen folgenden Kapiteln zu verbessern. Dabei wird ausschließlich die *„localVersion“* betrachtet. Der PIO-ULB Editor kann auf einer eigenen Webseite [3] ausprobiert werden. Diese Webseite zeigt jedoch die *„webVersion“* des Editors, weshalb geringfügige Abweichungen zu den hier abgebildeten Screenshots zu erwarten sind. Zum Entwicklungsprozess der Benutzeroberfläche wurde eine separates Paper veröffentlicht. [4]

Beim erstmaligen Öffnen des PIO-ULB Editors wird ein *„Disclaimer“* angezeigt, welcher darauf hinweist, dass der Editor nicht für die Nutzung im pflegerischen Alltag konzipiert wurde und es sich lediglich um einen Prototypen handelt. Außerdem wird der Nutzer über die Umfangsreduzierung der PIO-ULB Spezifikation (also das *PIO-Small*) informiert. Im Disclaimer kann ein Dokument heruntergeladen werden, welches die Inhalte des *PIO-Smalls* detailliert beschreibt. Alternativ ist das Dokument im Quellcode des Backends auf GitHub hinterlegt. Der *„Disclaimer“* muss akzeptiert werden, bevor der Nutzer den Editor verwenden kann.

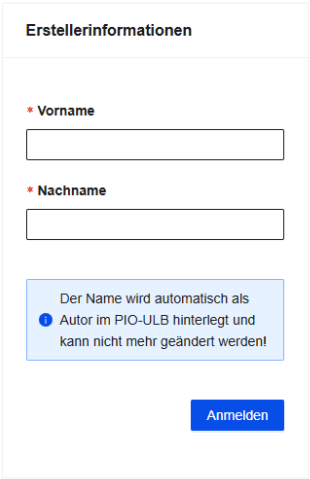

Abbildung 3: *Anmelde-Screen des PIO-ULB Editors*

Im nächsten Schritt muss sich der Nutzer mit seinem Vor- und Nachnamen anmelden (siehe Abb. 3). Ein robustes User Management mit Passwortvergabe gibt es in diesem Prototypen nicht. Sollte der Nutzer die Anwendung schließen, sind die Daten seiner Session weiterhin im Backend zeitbegrenzt gespeichert. Durch eine erneute Anmeldung mit demselben Namen kann auf die Session erneut zugegriffen werden. Außerdem wird der Name des Nutzers automatisch als Autor in jedem PIO-ULB hinterlegt, welches mit diesem Editor erstellt oder editiert wird.

[3] Technische Hochschule Augsburg, *PIO-ULB Editor*, URL: https://pio-editor.de/ [zuletzt aufgerufen: 10.03.2025].
[4] Mess EV, Balic S, Kleybolte L, Regner M, Werlitz V, et al., *Design und Entwicklung einer Nutzeroberfläche zur Darstellung des PIO-Überleitungsbogens*, In: Boll S, Hein A, et al., Hrsg. Zukunft der Pflege, *Tagungsband der 6. Clusterkonferenz 2023*, Oldenburg: University of Oldenburg Press, 2023, S. 30-34. URN: urn:nbn:de:gbv:715-oops-59467.

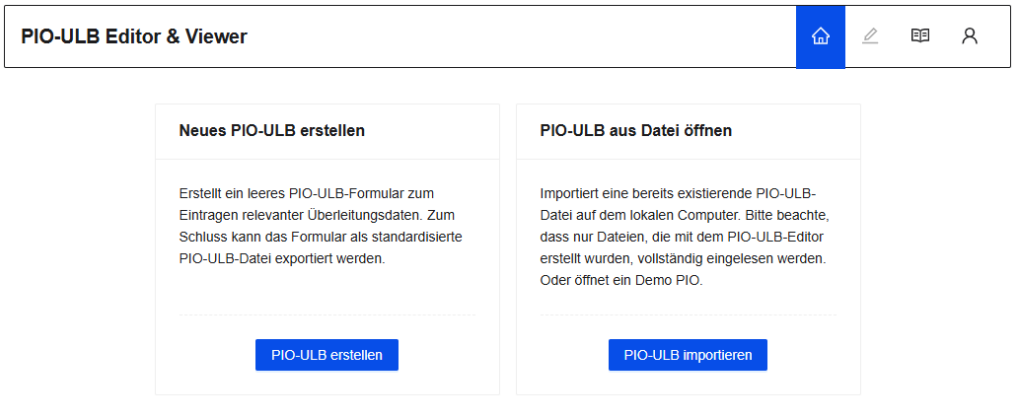

Abbildung 4: *Home-Screen des PIO-ULB Editors*

Nach der erfolgreichen Anmeldung wird der Home-Screen und ein Kopfzeilenmenü angezeigt (siehe Abb. 4). Das Kopfzeilenmenü ist nach der Anmeldung auf jedem weiteren Screen zu sehen, damit der Nutzer stets auf dessen Funktionen zugreifen kann. Das Menü stellt folgende Funktionen bereit:

- Springe zum Home-Screen (Haus Icon)
- Springe zur Editoransicht (Stift Icon)
- Springe zum Adressbuch-Screen (Buch Icon)
- Anzeige der Nutzerdaten inkl. Logout Button (Personen Icon)

Der Home-Screen stellt zwei Option zur Verfügung: Das Erstellen eines neuen PIO-ULBs und das Importieren eines PIO-ULBs. Die erste Option lädt die Editoransicht mit leeren Eingabefeldern, während die zweite Option ein Popup Fenster öffnet, welches die Auswahl einer Datei aus dem lokalen Dateisystem ermöglicht. Wenn die Datei ein valides PIO-ULB darstellt, werden die Daten ausgelesen und in der Editoransicht angezeigt.

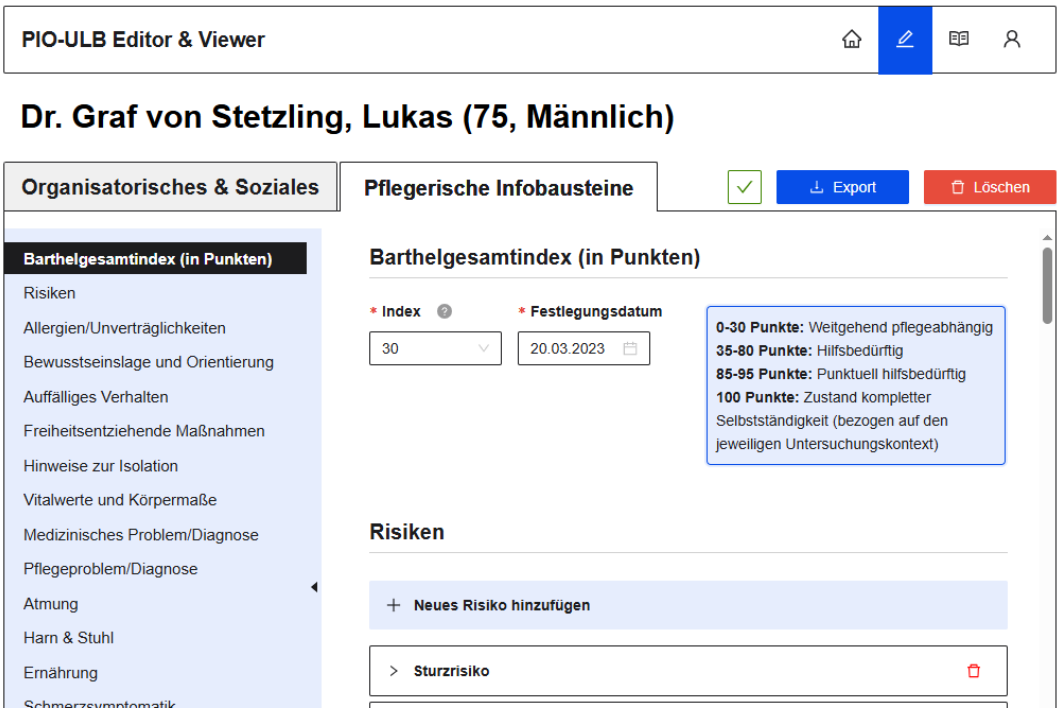

Abbildung 5: *Editoransicht des PIO-ULB Editors mit einklappbarem Seitenmenü (Tab: Pflegerische Infobausteine)*

Abbildung 5 zeigt die Editoransicht mit bereits befüllten Eingabefeldern. In dieser Ansicht können die Patientendaten eingetragen oder editiert werden. Die PIO-ULB Spezifikation gliedert die FHIR-Ressourcen in zwei Kategorien: „Organisatorisches & Soziales“ und „Pflegerische Informationsbausteine“. Diese Gliederung wurde ebenfalls im Editor umgesetzt. Abbildung 5 zeigt den Tab „Pflegerische Informationsbausteine“. Neben der Tab-Bar befindet sich das Tab-Menü, welches eine Löschen- und eine Exportfunktion bereitstellt. Das grüne Häkchen-Icon zeigt das Validierungsergebnis, welches in diesem Fall fehlerfrei ausgefallen ist. Der Validierungsprozess wird ausschließlich beim Import einer XML-Datei angestoßen. Durch einen Klick auf das Icon öffnet sich ein Fenster mit weiteren Details zum Validierungsergebnis.

Unterhalb der Tab-Bar können die Patientendaten eingetragen werden. Die Gliederung der Eingabefelder orientiert sich sehr stark an den FHIR-Ressourcen aus der PIO-ULB Spezifikation. Zum Beispiel gibt es in der Spezifikation eine eigene Ressource für die Abbildung des Barthelgesamtindexes sowie weitere Ressourcen, um Risiken darzustellen. Diese Struktur ist auch im PIO-ULB Editor (siehe Abb. 5) wiederzufinden. Da für jedes einzelnes Risiko eine eigene FHIR-Ressource vorgesehen ist, werden alle Risiko-Ressourcen als Akkordeonmenü dargestellt.

Abbildung 6: *Auswahlmöglichkeiten der Exportfunktion (XML-Export oder PDF-Export)*

Beim Klick auf den Export Button öffnet sich ein Popover mit zwei verschiedenen Exportmöglichkeiten. Der XML-Export generiert eine standardkonforme XML-Datei (also ein PIO-ULB), während der PDF-Export alle Patientendaten menschenlesbar in einer PDF-Datei abbildet. Die PDF-Datei spiegelt die UI-Komponenten des Editors unmittelbar wieder.

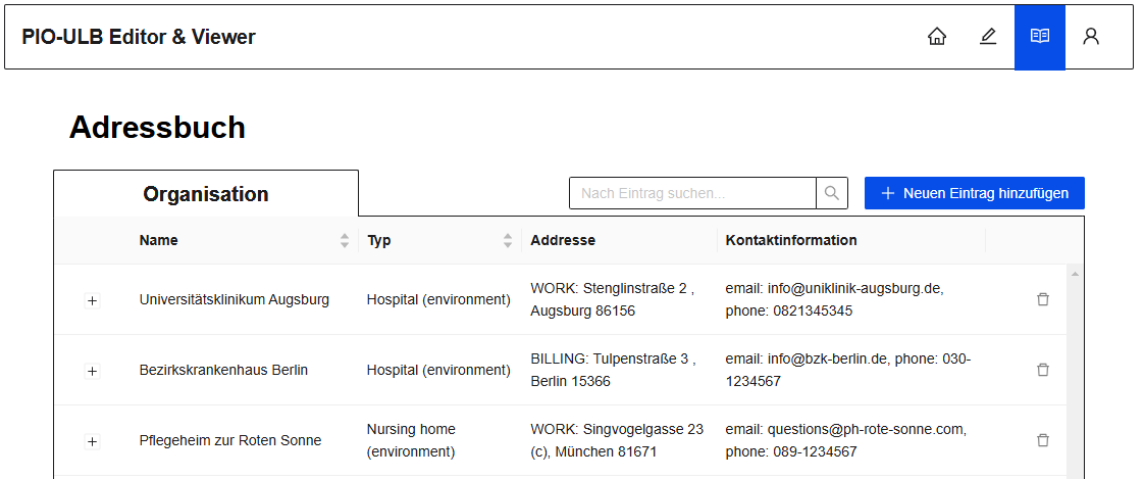

Abbildung 7: *Adressbuch-Screen des PIO-ULB Editors*

Das Adressbuch kann FHIR-Organisationsressourcen permanent in einer Datenbank speichern (siehe Abb. 7). Diese Organisationen können dann in der Editoransicht ausgewählt werden. So muss das Pflegepersonal nicht jedes Mal erneut Daten eines bekannten Pflegeheims oder Krankenhauses eintippen.

Abbildung 8: *Dritter Tab zur Anzeige von Daten, die nicht im PIO-Small enthalten sind*

Sollte ein PIO-ULB importiert werden, welches Daten enthält, die im *PIO-Small* nicht vorgesehen sind, können diese Daten nicht im ersten oder zweiten Tab angezeigt werden, weil keine entsprechenden Eingabefelder existieren. In diesem Fall wird ein dritter Tab eingeblendet (siehe Abb. 8), welcher diese Daten unstrukturiert visualisiert. Ein Editieren oder ein erneuter Export dieser Daten ist nicht möglich.

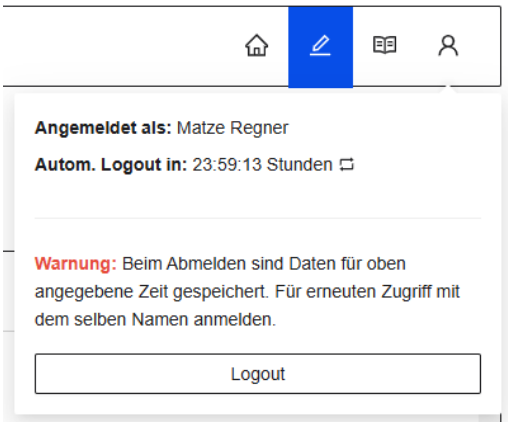

Abbildung 9: *Nutzerdaten im Kopfzeilenmenü inkl. Logout Button*

Das Popover, welches die Nutzerdaten im Kopfzeilenmenü anzeigt, informiert über den Nutzernamen und die Zeitspanne bis zum automatischen Löschen der Session, wobei alle eingetragenen Daten verloren gehen. Die Zeitspanne wird jedoch bei jeder Nutzerakkion zurückgesetzt. Ein vom Nutzer initiierter Logout oder das Schließen der Applikation führt nicht direkt zum Datenverlust. Die Daten bleiben für die angegebene Zeitspanne gespeichert. Ein erneuter Login mit demselben Namen innerhalb der Zeitspanne gewährt erneuten Zugriff auf die Daten.

4. Kontextabgrenzung

Das folgende Kontext-Diagramm ordnet den PIO-ULB Editor in den Prozess der Datenübermittlung im Rahmen einer Patientenüberleitung ein (siehe Abb. 10).

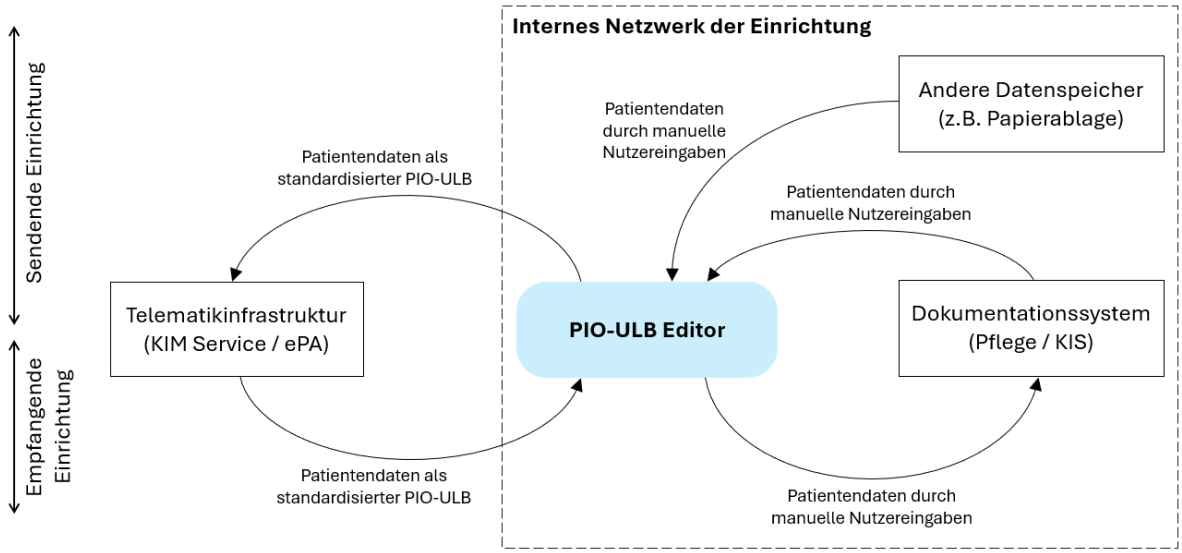

Abbildung 10: *Kontext-Diagramm des PIO-ULB Editors (obere Pfeile: entsendende Einrichtung; untere Pfeile: empfangende Einrichtung)*

Vor der Patientenüberleitung befinden sich die Patientendaten in unterschiedlichen Formen im Verwaltungsapparat der entsendenden Einrichtung und müssen letztendlich in das System der aufnehmenden Einrichtung übertragen werden. Im Fall eines Pflegeheimes werden pflegerische Dokumentationssysteme verwendet, im Krankenhaus speichert das Krankenhausinformationssystem (KIS) die Patientendaten. Der PIO-ULB Editor muss aus datenschutzrechtlichen Gründen im internen Netzwerk der Einrichtung betrieben werden, welche die Patientendaten verarbeitet. In Kapitel 2 wurde bereits erwähnt, dass der Editor eine „*localVersion*“ und eine „*webVersion*“ unterstützt (siehe Tabelle 4). Soll der Editor im lokalen Netzwerk der Einrichtungen laufen, ist hierfür die „*localVersion*“ vorgesehen, welche in dieser Dokumentation vorrangig betrachtet wird.

Der Prozess in der entsendenden Einrichtung wird durch die oberen Pfeile in Abbildung 10 dargestellt. Eine Pflegefachperson muss alle relevanten Patientendaten (evtl. aus verschiedenen Quellen) sammeln und manuell in den PIO-ULB Editor eintragen. Die aus dem Editor exportierte XML-Datei (= PIO-ULB) kann entweder in die elektronische Patientenakte (ePA) hochgeladen oder mittels KIM-Service (Kommunikation im Medizinwesen) versendet werden. Beide Optionen sind Services der Telematikinfrastruktur (TI). Die TI ist als deutsches Gesundheitsnetzwerk zu verstehen, über welches sensible Patientendaten datenschutzkonform versendet und gespeichert werden können. Im Rahmen des Forschungsprojektes wurden die PIO-ULBs über den KIM-Service versendet.

Der komplementäre Prozess in der aufnehmenden Einrichtung wird durch die unteren Pfeile in Abbildung 10 dargestellt. Die aufnehmende Einrichtung erhält den PIO-ULB in Form einer KIM-Nachricht. Alternativ kann der PIO-ULB auch aus der ePA heruntergeladen werden, nachdem der Patient entsprechende Zugriffsberechtigungen erteilt hat. Der

PIO-ULB wird dann in den Editor importiert. Die visualisierten Daten können anschließend am gleichen Computer manuell in das Dokumentationssystem der aufnehmenden Einrichtung übertragen werden.

5. Lösungsstrategien

Das folgende UML-Komponentendiagramm zeigt Top-Level-Entwurfsentscheidungen des PIO-ULB Editors (siehe Abb. 11). Alle Komponenten werden nachfolgend genauer erklärt.

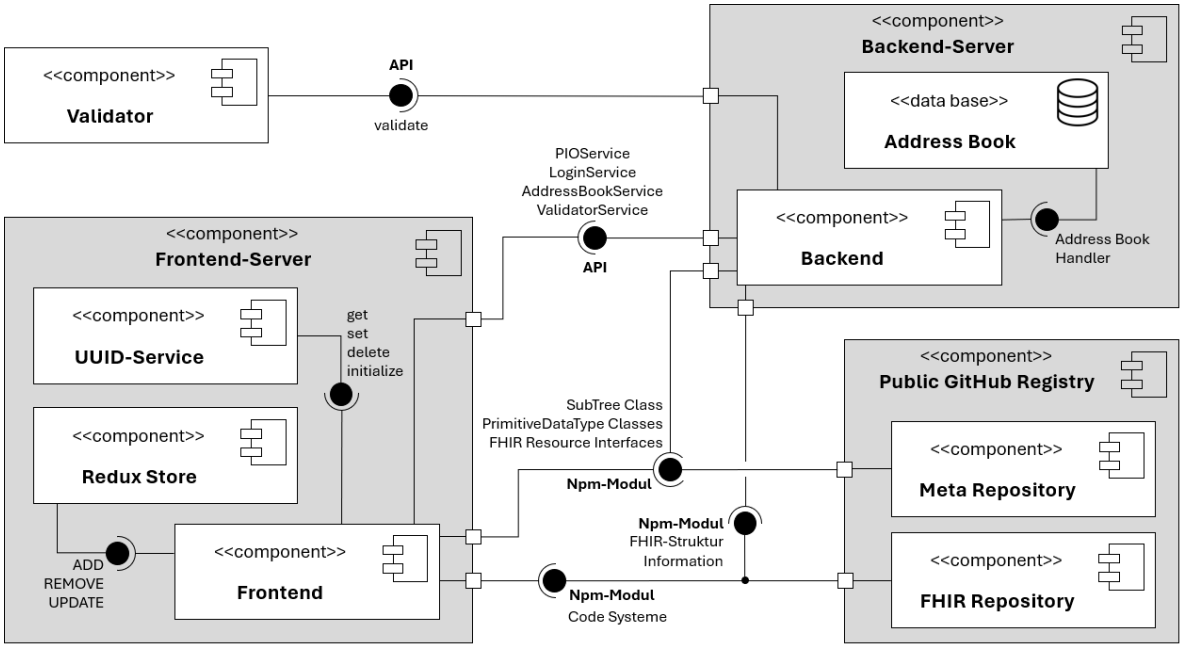

Abbildung 11: *UML-Komponentendiagramm des PIO-ULB Editors*

Der Frontend-Server sowie der Backend-Server sind als Hauptkomponenten zu verstehen, die jeweils Dienste des **Meta-** und **FHIR-Repositories** beanspruchen. Während das FHIR-Repository Strukturinformationen über den PIO-ULB sowie verwendete Codesysteme bereitstellt, bündelt das Meta-Repository Klassen, Typen und Interfaces, die in beiden Hauptkomponenten benötigt werden. Hierfür wurden eigene NPM-Module (Node Package Manager) implementiert, die in einer öffentlichen GitHub Registry [5] gehostet und dann eingebunden werden. Die **Validator-Komponente** wird vom Backend angesprochen und kann ein XML-Dokument offline validieren. Im Folgenden werden die Hauptkomponenten näher erläutert:

Der **Backend-Server** ermöglicht es mehreren Nutzern gleichzeitig, PIO-ULBs zu bearbeiten, insofern jeder Nutzer über ein eigenes Frontend angemeldet ist. Sobald die Nutzer ihren Vor- und Nachnamen eingegeben haben, erstellt der „LoginService“ eine neue Session. Die Eingabe eines Passwortes ist nicht nötig, da der Editor keine interne Nutzerverwaltung anbietet. Der Nutzername wird im PIO-ULB automatisch als Autor hinterlegt. Für jede Session werden alle weiteren Nutzereingaben zeitbegrenzt serverseitig gespeichert, sodass durch einen Absturz bzw. ein Beenden des Frontends keine Daten verloren gehen. Durch ein erneutes Anmelden mit demselben Namen, kann wieder auf die zuvor geöffnete Session zugegriffen werden. Melden sich zwei Nutzer mit identischem Namen an und bearbeiten gleichzeitig ein PIO-ULB, kommt es zu ungewollten Nebeneffekten, da beide Nutzer auf derselben Datenbasis arbeiten. Eine korrekte Abwicklung dieses Use-

[5] GitHub, *Technische Hochschule Augsburg – Institut für agile Softwareentwicklung*, URL: https://github.com/THAias [zuletzt aufgerufen: 10.03.2025].

Cases ist für die prototypische Implementierung nicht nötig und soll perspektivisch vom User Management System der jeweiligen Einrichtung übernommen werden. Weiterhin stellt das Backend Logik zum Importieren und Exportieren von XML-Dateien bereit. Eine globale serverseitige Datenbank – das Adressbuch – kann FHIR-Organisations-Ressourcen permanent speichern, sodass die Nutzer nicht jedes Mal erneut Daten eines bekannten Pflegeheimes oder Krankenhauses eingeben müssen.

Der **Frontend-Server** ist als React-Webanwendung implementiert und kann über den Browser aufgerufen werden. Die Benutzeroberfläche verwendet vorgefertigte Interface-Komponenten aus der „Ant Design“ Bibliothek für die Dateneingabe. Thematisch zusammengehörende Eingabefelder sind zu einem Formular zusammengefasst. Dies ermöglicht eine selbstständige Datenverarbeitung durch jedes Formular selbst (siehe Kapitel 6.4). Um jede FHIR-Ressource eindeutig zu identifizieren, wird ein „Universally Unique Identifier“ (UUID) verwendet. Die UUIDs von jeder zur Laufzeit erstellten FHIR-Ressource werden vom UUID-Service gespeichert, damit eingegebene Daten stets der richtigen Ressource zugeordnet werden können. Der Redux Store wird als Zustandsspeicher verwendet. Hier sind Nutzer- und Navigationsinformationen sowie Daten, die über mehrere React-Komponenten synchron gehalten werden müssen, temporär abgelegt.

6. Bausteinsicht

Dieses Kapitel befasst sich mit der Abstraktion des Sourcecodes, indem alle fünf Komponenten aus dem Komponentendiagramm (siehe Abb. 11) detaillierter beschrieben werden. Der PIO-ULB Editor setzt sich aus folgenden fünf Komponenten bzw. **Subsystemen** zusammen:

- Meta-Repository
- FHIR-Repository
- Backend-Server
- Frontend-Server
- Validator Service

Nachfolgende Unterkapitel befassen sich mit der Struktur dieser fünf Subsysteme. Jedes Subsystem wird bis auf Dateiebene aufgeschlüsselt. Am Schluss des Kapitels soll der Leser den Nutzen und die Funktion jeder einzelnen Datei verstanden haben, ohne dabei auf den Quellcode, also den Inhalt der Dateien, eingegangen zu sein. Der Quellcode für alle Subsysteme ist auf GitHub veröffentlicht [6].

6.1 Subsystem: Meta-Repository

Das Meta-Repository ist ein eigenes node-Projekt, welches in *TypeScript* geschrieben ist. Es wird im Frontend sowie im Backend eingebunden und stellt Code bereit, der in mehreren Subsystemen benötigt wird. Abbildung 12 zeigt schematisch die Bestandteile des Meta-Repositories:

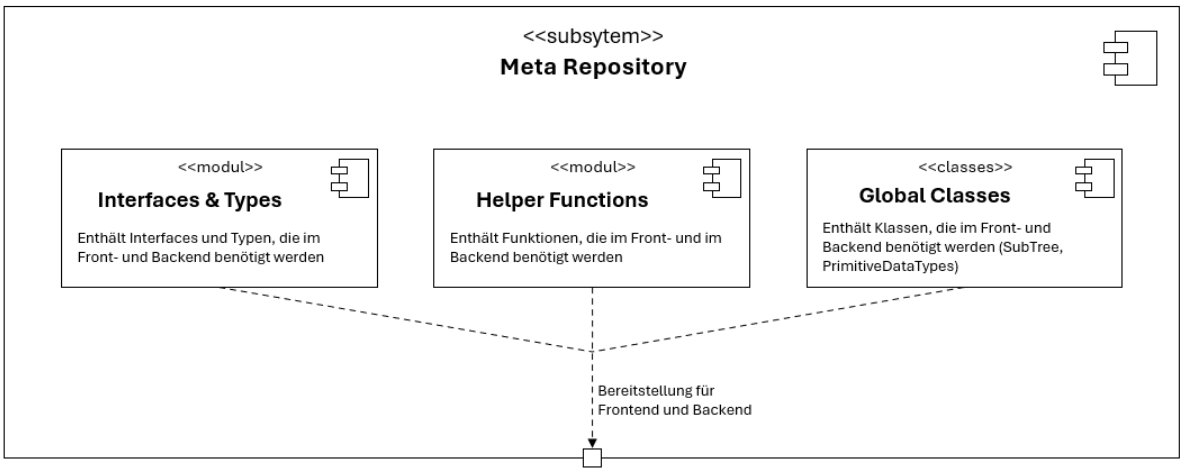

Abbildung 12: *Bestandteile des Subsystems „Meta-Repository“ und deren Zusammenspiel*

Das Modul „Interfaces & Types“ bündelt alle Interfaces und Typen, die sowohl im Frontend als auch im Backend benötigt werden. Interessant sind hierbei die „ObjectTypes“ (siehe Tab. 1). Es handelt sich um Interfaces, welche alle Daten einer FHIR-Ressource in einer flachen Hierarchie darstellen. Zum Beispiel hält das Interface

[6] GitHub, *Technische Hochschule Augsburg – Institut für agile Softwareentwicklung*, URL: https://github.com/THAias [zuletzt aufgerufen: 10.03.2025].

„IMedicalProblemObject“ alle Daten einer FHIR-Ressource, die eine medizinische Diagnose abbildet. Diese Interfaces wurden für alle FHIR-Ressourcen angelegt, die in mehrfacher Anzahl im PIO-ULB vorkommen können, und deshalb im Frontend in Form von Akkordeon Menüs dargestellt werden. So existiert zum Beispiel im Frontend für jede medizinische Diagnose des Patienten ein „IMedicalProblemObject“, welche innerhalb eines Akkordeon Menüs visualisiert werden.

Das Modul „Global Classes“ stellt einige Klassendefinitionen für das Frontend und das Backend bereit. Die 12 PrimitiveDataType-Klassen bilden die für den PIO-ULB relevanten Datentypen ab, die vom FHIR Standard vorgesehen sind (z.B. String, Code, Uri, Date, DateTime, Integer, usw.). Informationen, die der Nutzer in den Editor einträgt, werden als Instanz der PrimitiveDataType-Klassen abgespeichert. Weiterhin umfasst das Modul die SubTree-Klasse, welche zwischen Front- und Backend verschickt wird, um Daten aus dem PIO-ULB auszutauschen. Ein SubTree ist ein rekursiv aufgebautes Objekt, welches einen Teilausschnitt aus der XML-Struktur eines PIO-ULBs abbildet (siehe Kapitel 9.2).

Logik, die das Frontend und das Backend benötigen, werden als „Helper Functions“ bereitgestellt. Zum einen findet man Funktionen zur Verarbeitung von Zeit- und Datumsinformationen, zum anderen Funktionen zur Generierung und Manipulation von SubTree Instanzen (siehe Kapitel 12).

Zum Schluss listet Tabelle 1 alle Dateien auf, die im Quellcode des Meta-Repositories enthalten sind, und ordnet diese einem Modul aus Abbildung 12 zu.

Tabelle 1: *Auflistung und Erklärung aller Dateien aus dem Quellcode des Meta-Repositories*

Modul	Datei	Beschreibung
Interfaces & Types	@types/DataTypesInterfaces.ts	Verschiedene Interfaces und Typen
	@types/Types.ts	
	@types/ObjectTypes.ts	Interfaces mit flacher Hierarchie, die eine gesamte FHIR-Ressource abbilden und im Frontend durch Akkordeon Menüs visualisiert werden.
	@types/ResponseInterface.ts	Interface, welches als Antwort auf eine HTTP-Request gesendet wird.
	@types/UserDataInterface.ts	Interface zur Abbildung von Benutzerdaten
Helper Functions	Helper.ts	Funktionen, die in Frontend und Backend benötigt werden.
Global Classes	PrimitiveDataTypes.ts	12 Klassen, die von FHIR vorgesehene Datentypen abbilden (z.B. String, DateTime, Uri, Integer, usw.).
	SubTree.ts	Klasse zur Kommunikation zwischen Frontend und Backend

6.2 Subsystem: FHIR-Repository

Das FHIR-Repository ist ebenfalls ein eigenes node-Projekt, das in *TypeScript* geschrieben ist. Dieses Subsystem enthält Quellcode zur Generierung von LookUpTables, welche FHIR-Strukturinformationen über den PIO-ULB sowie alle verwendeten Codesysteme inklusive Codes und deren Bedeutung bereitstellen. Die FHIR-Strukturinformationen

werden dem Backend bereitgestellt, während die Codesysteme fast ausschließlich in Drop Down Menüs im Frontend verwendet werden. Abbildung 13 zeigt schematisch die Bestandteile des FHIR-Respositories:

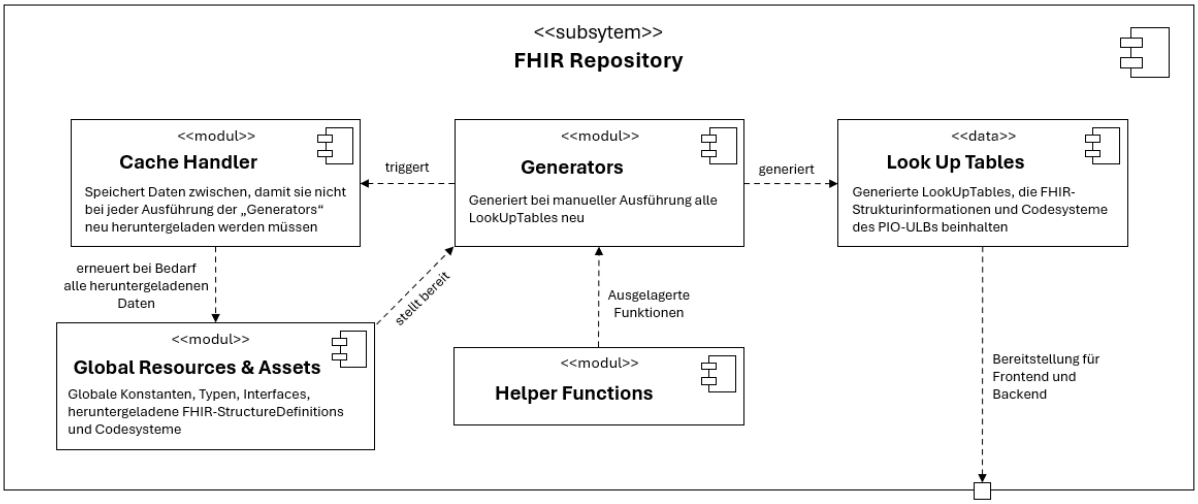

Abbildung 13: *Bestandteile des Subsystems „FHIR-Repository“ und deren Zusammenspiel*

Das Modul „Generators“ besteht aus mehreren Dateien, die für die Erstellung von vier LookUpTables zuständig sind. Die Generatoren müssen manuell durch die Ausführung der Datei *„main.ts“* gestartet werden. Eine automatische Ausführung durch die Software selbst ist nicht vorgesehen. Während der Generierung werden sowohl FHIR-Strukturdefinitionen als auch gesamte Codesysteme und deren deutsche Übersetzung heruntergeladen. FHIR-Strukturdefinitionen sind XML-Dateien, die den Aufbau einer FHIR-Ressource exakt beschreiben. Leider sind deutsche Übersetzungen nicht für alle Codes verfügbar, weshalb einige Codes in englischer Sprache angezeigt werden. Die meisten Informationen über den PIO-ULB werden von der Mio42 GmbH als eigenes NPM-Modul bereitgestellt. [7] Hierbei handelt es sich um FHIR-Strukturdefinitionen, Codesysteme und sogenannte ConceptMaps, die deutsche Übersetzungen für einige Codesysteme enthalten. Diese Dateien werden im *„node_modules“* Ordner gespeichert, wie es für alle NPM-Module üblich ist. Das NPM-Modul beinhaltet jedoch für den PIO-ULB Editor unbrauchbare FHIR-Strukturdefinitionen, weil diese nur die Unterschiede zu den FHIR-Core-Ressourcen auflisten, der PIO-ULB Editor jedoch die *„snapshot“*-Variante benötigt. Diese Variante bildet die gesamte Struktur einer FHIR-Ressource ab und muss von simplifier.net [8] heruntergeladen werden. Weitere deutsche Übersetzungen für verwendete Codesysteme werden vom SNOMED-CT Browser [9] heruntergeladen.

Da durch die zahlreichen Downloadvorgänge beim mehrmaligen Starten der Generatoren die IP-Adresse des verwendeten Computers gesperrt wird, werden alle heruntergeladenen Daten im „Cache Handler“ für vier Wochen gespeichert. Erst nach Ablauf der Zeit werden die Daten bei einer wiederholten Ausführung der Generatoren neu

[7] HL7 FHIR Package Registry, *kbv.mio.ueberleitungsbogen*, URL: https://registry.fhir.org/package/kbv.mio.ueberleitungsbogen%7C1.0.0 [zuletzt aufgerufen: 11.03.2025].

[8] Simplifier.net, *PIO Überleitungsbogen V1.0.0*, URL: https://simplifier.net/ulb [zuletzt aufgerufen: 11.03.2025].

[9] SNOMED International, *SNOMED CT Browser*, URL: https://browser.ihtsdotools.org/?perspective=full&conceptId1=404684003&edition=MAIN/2025-03-01&release=&languages=de,en [zuletzt aufgerufen 11.03.2025].

heruntergeladen. Da eine häufige Änderung der Quelldaten sehr unwahrscheinlich ist, stellt die Verwendung eines Cache Systems kein Problem dar.

Das Modul „Global Resources & Assets“ stellt globale Konstanten, Typen und Interfaces bereit, die von mehreren Dateien im Subsystem benötigt werden. Außerdem werden hier die heruntergeladenen FHIR-Strukturdefinitionen und Codesysteme gespeichert und manchmal vom „Cache Handler“ überschrieben. Sowohl die FHIR-Strukturdefinitionen als auch die Codesysteme werden als JSON-Datei gespeichert. Letztendlich greift das Modul „Generators“ auf diese Daten zu und generiert die LookUpTables. Insgesamt werden vier LookUpTables zur Verwendung in anderen Subsystemen exportiert (siehe Tab. 2).

Tabelle 2: *Erläuterungen zu den LookUpTables des FHIR-Repositories*

LookUpTable	Beschreibung
ResourceLookUpTable.json	Die Datei wird vom Generator-Modul erstellt und listet **alle** im PIO-ULB vorkommenden FHIR-Ressourcen auf. Für jede Ressource sind die Profile-URL und alle möglichen XML-Pfade, an denen Daten gespeichert werden können, hinterlegt. Für jeden XML-Pfad ist der primitive Datentyp, der an dieser Stelle erwartet wird, angegeben. Die Information welcher Datentyp an welcher Stelle in der XML-Datei zu erwarten ist, wird für die Deserialisierungslogik im Backend benötigt. Falls vorhanden, werden Codesysteme oder fixe Werte, die vom PIO-ULB Standard vorgegeben sind, für entsprechende XML-Pfade aufgeführt.
PioSmallLookUpTable.json	Die Datei wird vom Generator erstellt und listet nur die FHIR-Ressourcen und XML-Pfade auf, die im **PIO-Small** enthalten sind. Demnach ist diese Datei eine gekürzte Version der ResourceLookUpTable.
ValueSetLookUpTable.json	Die Datei wird vom Generator erstellt und listet **alle** im PIO-ULB vorkommenden **Codesysteme** auf. Für jeden Code ist das zugrundeliegende Codesystem und dessen Versionsnummer sowie ein Anzeigename hinterlegt.

```
"KBV_PR_MIO_ULB_Device_Implant": {
    "resource": {
        "status": "extensions",
        "profile": "https://fhir.kbv.de/StructureDefinition/KBV_PR_MIO_ULB_Device_Implant|1.0.0",
        "fhir-resource-type": "Device"
    },
    "paths": {
        "KBV_PR_MIO_ULB_Device_Implant.type.coding:snomedCode": {
            "type": "CodingPIO",
            "valueSet": "https://fhir.kbv.de/ValueSet/KBV_VS_MIO_ULB_Medical_Device|1.0.0"
        },
```

Abbildung 14: *Auszug aus der ResourceLookUpTable.json*

```
"https://fhir.kbv.de/ValueSet/KBV_VS_MIO_ULB_Level_of_Care": [
    {
        "system": "http://fhir.de/CodeSystem/bfarm/ops",
        "version": "2021",
        "code": "9-984.6",
        "display": "Pflegebedürftig nach Pflegegrad 1"
    },
    {
```

Abbildung 15: *Auszug aus der ValueSetLookUpTable.json*

TranslationListOf ExcludedPaths.json	Die Datei wird vom Generator erstellt und enthält für alle ausgeschlossenen XML-Pfade eine deutsche Bezeichnung, die für die Weiterverwendung im Subsystem „Frontend“ exportiert werden. Wird ein PIO-ULB im Editor importiert, welcher aus dem *PIO-Small* ausgeschlossene Daten beinhaltet, können diese im Frontend nicht wie gewöhnlich visualisiert werden, sondern müssen als Liste in einem dritten Tab (siehe Abb. 8) angezeigt werden. Dieser dritte Tab bedient sich der deutschen Bezeichnungen aus der TranslationListOfExcludedPaths, um die Liste lesbarer zu gestalten.

```
"KBV_PR_MIO_ULB_Encounter_Hospital_Stay": {
    "wholeResourceExcluded": true,
    "translation": "Letzter Krankenhausaufenthalt",
    "cardinalityReducedToOne": [],
    "excludedPaths": {
        "KBV_PR_MIO_ULB_Encounter_Hospital_Stay.period.end": "Enddatum"
    }
```

Abbildung 16: *Auszug aus der TranslationListOfExcludedPaths.json*

Zum Schluss listet Tabelle 3 alle Dateien auf, die im Quellcode des FHIR-Repositories enthalten sind, und ordnet diese einem Modul aus Abbildung 13 zu.

Tabelle 3: *Auflistung und Erklärung aller Dateien aus dem Quellcode des FHIR-Repositories*

Modul	Ordner / Datei	Beschreibung
Generators	generators/main.ts	Diese Datei muss manuell ausgeführt werden, um die Generierung der LookUpTables zu starten. Eine erneute Generierung ist nötig, wenn eine neue Version des PIO-ULB Standards veröffentlicht wurde.
	generators/GermanRefSet-Downloader.ts	Herunterladen der deutschen Übersetzungen für einige Codesysteme
	generators/GermanTranslation-LookUpTable.ts	
	generators/ResourcesLookUpTables.ts	Generierung der ResourceLookUpTable
	generators/PioSmallLookUpTable.ts	Generierung der PioSmallLookUpTable durch Herauslöschen von ausgeschlossenen XML-Pfaden aus der ResourceLookUpTable
	generators/TransformXML-Resources.ts	Wandelt die FHIR-Strukturdefinitionen vom XML-Format in das JSON-Format.
	generators/ValueSetLookUpTable.ts	Generierung der ValueSetLookUpTable
Cache Handler	Cache Handler.ts	Kümmert sich um das Zwischenspeichern der FHIR-Strukturdefinitionen und Codesysteme für vier Wochen, damit diese nicht bei jeder Ausführung der Generatoren erneut heruntergeladen werden müssen.
Global Ressources & Assets	@types	Im Ordner „@types“ befinden sich Typen und Interfaces, die für die Generatoren benötigt werden.
	helper/PioSmallExclusions.json	Diese Datei wurde manuell erstellt und enthält alle XML-Pfade, die nicht Teil des *PIO-Smalls* sind. Für alle ausgeschlossenen Pfade ist eine deutsche Bezeichnung hinterlegt. Sie dient als Grundlage für die Generierung der PioSmallLookUpTable.json und der TranslationListOfExcludedPaths.json.

	../node_modules/ kbv.mio.ueberleitungsbogen	In diesem Ordner sind alle FHIR-Strukturdefinitionen, Codesysteme und ConceptMaps des PIO-ULB gespeichert, die in dem NPM-Modul der Mio42 GmbH enthalten sind.
	../.cache	In diesem Ordner sind alle FHIR-Strukturdefinitionen als *„snapshot“*-Variante sowie alle deutschen Übersetzungen des SNOMED-CT Browsers zwischengespeichert.
Helper Functions	Helper.ts	Funktionen, die in mehreren Dateien des FHIR-Repositories Anwendung finden.
Look Up Tables	data	Im Ordner „data“ befinden sich alle LookUpTables die exportiert werden, um in anderen Subsystemen weiterverwendet zu werden. Der Export dieser Tables findet in der Datei „index.ts“ statt. Eine detaillierte Erklärung zu allen LookUpTables ist in Tabelle 2 zu finden.

6.3 Subsystem: Backend-Server

Das Backend ist ein eigenes node-Projekt, welches in *TypeScript* programmiert ist. Das Herzstück dieses Subsystems ist die Serialisierungs- bzw. Deserialisierungslogik zum Erstellen bzw. Einlesen von XML-Dateien (= PIO-ULB). Weiterhin kümmert sich das Backend um das Session-Management und das permanente Speichern von FHIR-Organisations-Ressourcen in einer Datenbank. Das Subsystem kann lokal durch das start-Skript in der *package.json* gestartet werden. Abbildung 17 zeigt schematisch die Bestandteile des Backends:

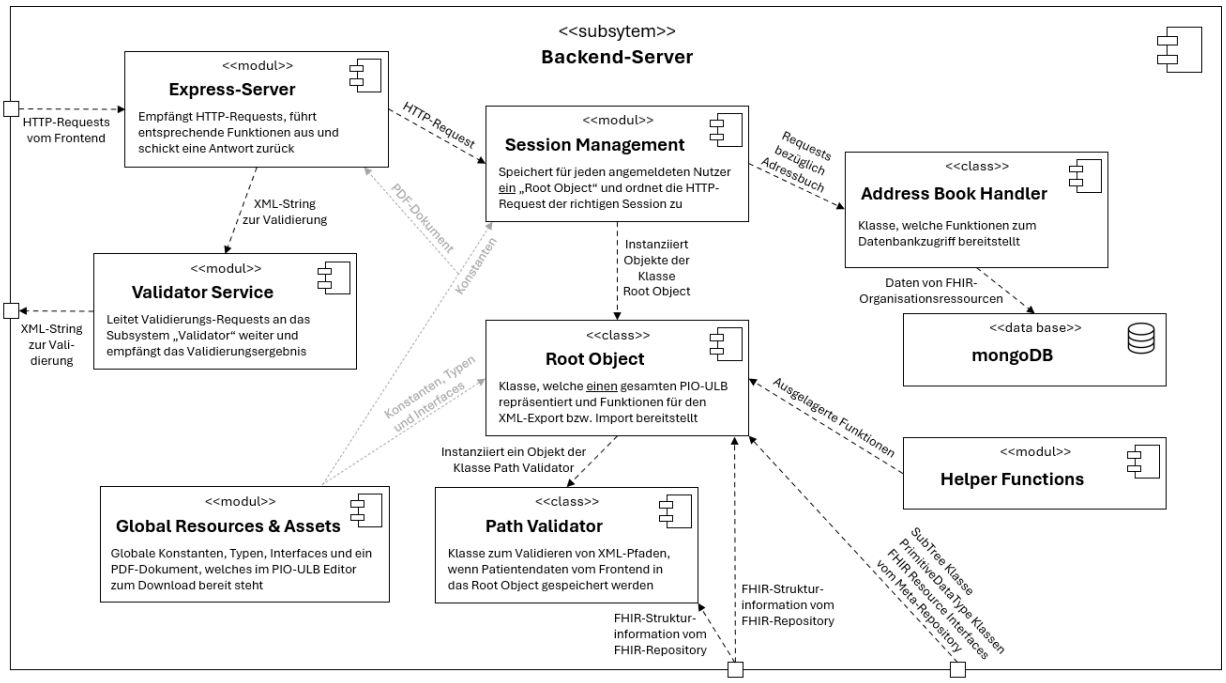

Abbildung 17: *Bestandteile des Subsystems „Backend“ und deren Zusammenspiel*

Der „Express Server“ stellt eine API Schnittstelle (Application Programming Interface) bereit, über welche alle HTTP-Anfragen vom Frontend abgewickelt werden. Wenn die Anfrage einen Validierungsauftrag enthält, wird der empfangene XML-String an den „Validator Service“ (siehe Kapitel 6.5) weitergeleitet. In allen anderen Fällen wird die Anfrage vom Backend selbst bearbeitet. Diese Architektur ermöglicht die Bündelung aller HTTP-Anfragen, die vom Frontend gesendet werden, auf eine Schnittstelle, wodurch weniger Abhängigkeiten und somit weniger Synchronisationsaufwand entstehen.

HTTP-Anfragen, die das Backend selbst bearbeitet, müssen als erstes einer Session zugeordnet werden, da mehrere Frontend-Clients und damit auch mehrere Nutzer gleichzeitig auf dasselbe Backend zugreifen können. Diese Aufgabe übernimmt das Modul „Session Management“. Beim erstmaligen Anmelden eines Nutzers wird ein JSON-Web Token erstellt, welcher im Frontend hinterlegt und bei jeder weiteren Anfrage zur Identifizierung der Session verwendet wird. Das Verhalten des „Session Managements“ ist abhängig davon, wie das Backend gestartet wurde. Die Umgebungsvariable *VERSION_ENV* existiert sowohl im Frontend als auch im Backend und steuert, ob der PIO-ULB Editor in der *„localVersion“* oder *„webVersion“* gestartet wird. Tabelle 4 erläutert den Unterschied zwischen den beiden Versionen:

Tabelle 4: *Erläuterung der "localVersion" und "webVersion" (gesteuert durch die Umgebungsvariable VERSION_ENV im Frontend und im Backend)*

Version	Erklärung
localVersion	Diese Version ist für den lokalen Betrieb im Netzwerk von **medizinischen und pflegerischen Einrichtungen** gedacht. Das Verhalten der Software wird wie folgt beeinflusst: • Nach einer Abmeldung oder Schließung des Browserfensters bleiben die Session-Daten für 24 Stunden gespeichert • Automatischer Logout nach 24 Stunden Inaktivität • Nutzer können auf ihre Session erneut zugreifen, indem sie sich mit demselben Vor- und Nachnamen anmelden • Melden sich zwei Nutzer mit demselben Vor- und Nachnamen an, kommt es zu unerwünschten Komplikationen • Das Adressbuch aller Clients greift auf die Datenbank im Backend zu, in der FHIR-Organisations-Ressourcen global gespeichert werden • Keine Bereitstellung eines Demo-PIO-ULBs
webVersion	Diese Version ist für den Betrieb auf einer **öffentlichen Website** gedacht, auf der der PIO-ULB Editor zu Demonstrationszwecken gehostet wird. Das Verhalten der Software wird wie folgt beeinflusst: • Nach einer Abmeldung oder Schließung des Browserfensters werden die Session-Daten sofort gelöscht • Automatischer Logout nach 4 Stunden Inaktivität • Für Nutzer, die sich gleichzeitig mit demselben Vor- und Nachnamen anmelden, wird jeweils eine separate Session erstellt • Das Adressbuch aller Clients greift nicht auf die Datenbank im Backend zu, sondern auf vorinitialisierte Testdaten im Frontend • Bereitstellung eines Demo-PIO-ULBs zu Demonstrationszwecken

Wie in Tabelle 4 ersichtlich wird in der *„webVersion“* für jede Anmeldung eine eigene Session erstellt, auch wenn der Vor- und Nachname zweier Nutzer identisch ist. Dies funktioniert über einen einzigartigen *Fingerprint*, der für jeden Frontend-Client neu generiert wird. Öffnet ein Nutzer den PIO-ULB Editor in zwei verschiedenen Browser-Tabs und

meldet sich mit demselben Namen an, wird trotzdem für jeden Tab eine separate Session angelegt, da in jedem Browser-Tab ein eigener *Fingerprint* erstellt wird.

HTTP-Anfragen, die das Adressbuch betreffen, werden an das Modul „Address Book Handler“ weitergeleitet (siehe Abb. 17). Dieses Modul stellt Funktionen zum Erstellen, Editieren, Löschen und Auslesen von Datenbankeinträgen bereit. Um die Adressbucheinträge bzw. die FHIR-Organisations-Ressourcen persistent zu speichern, wird eine mongoDB Datenbank verwendet. [10] Die Datenbank wird als Docker-Image heruntergeladen und gestartet.

Das Herzstück des Backends ist die „Root Object“ Klasse, welche alle Daten eines PIO-ULBs speichert und die Serialisierungs- bzw. Deserialisierungslogik umfasst. Für jede Session wird eine Instanz dieser Klasse angelegt. Dies bedeutet, dass in einer Session maximal ein PIO-ULB gleichzeitig bearbeitet werden kann. Abbildung 18 zeigt ein gekürztes Klassendiagramm des „Root Objects“:

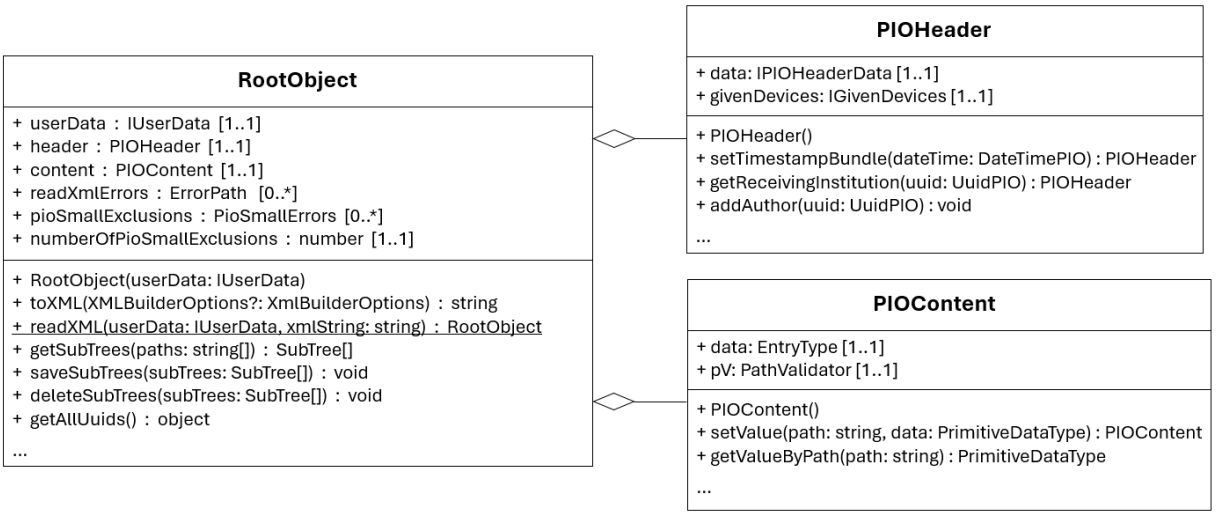

Abbildung 18: *Gekürztes UML-Klassendiagramm vom "Root Object"*

Die Attribute *„header“* und *„content“* halten jeweils eine Instanz einer anderen Klasse, welche alle Header-Daten eines PIO-ULBs bzw. alle inhaltlichen Daten eines PIO-ULBs speichert. Als Header-Daten werden alle Informationen des FHIR-Bundles und der FHIR-Composition angesehen.

Die Attribute *„readXmlErrors“* und *„pioSmallExclusions“* sind für den Import eines PIO-ULBs relevant. Ersteres speichert alle XML-Pfade, die nicht in der offiziellen PIO-ULB Spezifikation vorgesehen sind und somit vom Standard abweichen. Diese Elemente kann der PIO-ULB Editor nicht anzeigen. Letzteres sichert alle XML-Pfade, die Teil der offiziellen PIO-ULB Spezifikation sind, aber nicht Teil des *PIO-Smalls*. Der PIO-ULB Editor kann nur Elemente des *PIO-Smalls* vollumfänglich visualisieren und editieren. Alle ausgeschlossenen Elemente werden in *„pioSmallExclusions“* hinterlegt und unstrukturiert in einem dritten Tab (siehe Abb. 8) im Frontend angezeigt. Die Detektion dieser XML-Pfade funktioniert

[10] MongoDB, *Was ist MongoDB*, URL: https://www.mongodb.com/de-de/company/what-is-mongodb [zuletzt aufgerufen: 11.03.2025].

anhand der FHIR-Strukturinformationen, die vom Subsystem FHIR-Repository bereitgestellt werden.

Die *„toXML“* Methode der „Root Object“ Klasse generiert aus allen hinterlegten Daten einen XML-String, während die statische *„readXML“* Methode einen XML-String entgegennimmt und alle ausgelesenen Informationen im „Root Object“ speichert. Eine genaue Beschreibung dieser Serialisierungs- bzw. Deserialisierungslogik befindet sich in Kapitel 7.6 bzw. 7.7.

Weiterhin gibt es Methoden zum Abrufen, Speichern und Löschen von SubTrees. Ein SubTree stellt einen beliebigen Ausschnitt aus der XML-Baumstruktur eines PIO-ULBs dar und wird für die Kommunikation zwischen Frontend und Backend verwendet (siehe Kapitel 7.4 & 9.2). Mit Hilfe der SubTrees werden Nutzereingaben in das „Root Object“ übertragen.

Die Methode *„getAllUuids()“* gibt alle bereits verwendeten UUIDs zurück und ordnet jede einzelne einer FHIR-Ressource zu. Jede FHIR-Ressource ist mit einer einzigartigen UUID eindeutig identifizierbar.

Als nächstes soll die Klasse *„PIOContent“* näher betrachtet werden (siehe Abb. 18). Diese Klasse speichert alle inhaltlichen Daten eines PIO-ULBs und greift dabei auf eine spezielle Struktur zurück, die durch den Typ *„EntryType“* definiert wird. Dieser Typ wird vom Subsystem Meta-Repository bereitgestellt und ist in Abbildung 19 am Beispiel der Patientenadresse (siehe Abb. 2) dargestellt.

```
{ "24ed8a3b-59f7-47fd-8699-bb908de982c0": {
      "KBV_PR_MIO_ULB_Patient": {
            "…": { … },
            "address": [ {
                  "type": {
                        "_ _value": CodePIO(„postal") },
                  "use": {
                        "_ _value": CodePIO("home") },
                  "line": [ {
                        "_ _value": StringPIO("Tulpenweg 14, c"),
                        "extension": [ {
                              "_ _url": UriPIO("http://hl7.org/fhir/StructureDefinition/iso21090-ADXP-streetName"),
                              "valueString": {
                                    "_ _value": StringPIO("Tulpenweg") } },
                        { … }, { … } ] } ],
                  "postalCode": {
                        "_ _value": StringPIO("86150") },
                  "city": {
                        "_ _value": StringPIO("Augsburg") },
                  "text": {
                        "_ _value": StringPIO("Tulpenweg 14, c, 86150 Augsburg") }
            } ],
            "…": { … },
} } }  as  EntryType
```

Abbildung 19: *Schematische Darstellung der Patientenadresse als "EntryType" (grün: primitive Daten, blau: Pfad bis zur Postleitzahl)*

Das gesamte FHIR-Bundle wird ressourcenweise als „EntryType“ (also als JSON-Objekt) im Backend gehalten, wobei auf Kompatibilität zum verwendeten XML-Parser (fast-xml-parser, NPM-Modul) [11] geachtet wurde. Die Composition (das Inhaltsverzeichnis) ist zwar eine eigene FHIR-Ressource, wird aber nicht als „EntryType“ abgebildet, da sie bei einem

[11] Bibliothek zum Generieren und Einlesen von XML-Strings

Export automatisch generiert werden kann. Die Inhalte aller anderen Ressourcen werden als „EntryType“ gespeichert, wobei die Baumstruktur stets konform zur PIO-ULB Spezifikation gehalten wird.

Vom Nutzer eingetragene Daten werden als *„PrimitiveDataType“* am Ende eines Baumpfades abgelegt (grüne Markierung in Abb. 19). Die *„PrimitiveDataTypes“* sind speziell für den Editor entwickelte Klassen und werden vom Subsystem Meta-Repository bereitgestellt. An welcher Stelle der Baumstruktur welcher Datentyp verwendet wird, ist in der PIO-ULB Spezifikation genau definiert. In einem PIO-ULB sind alle Daten als String in der XML-Datei hinterlegt. Somit fehlt beim Import eines PIO-ULBs die Information, welcher Datentyp ursprünglich zugrunde lag. Damit die Deserialisierungslogik die XML-Datei in einen *„EntryType“* konvertieren kann, werden also strukturbezogene Datentypinformationen benötigt, die durch das FHIR-Repository mit Hilfe der ResourceLookUpTable bereitgestellt werden.

Weiterhin ist das Backend möglichst **generisch** implementiert. Dies bedeutet, dass das Backend lediglich Daten in eine Baumstruktur schreibt, ohne die Struktur zu validieren. Das Frontend muss also den Pfad kennen, unter dem die PIO-ULB Daten laut Spezifikation abgespeichert werden sollen. Um z.B. auf die Postleitzahl der hinterlegten Patientenadresse zuzugreifen, muss folgender absoluter Pfad verwendet werden (vgl. Abb. 19):

24ed8a3b-59f7-47fd-8699-bb908de982c0.KBV_PR_MIO_ULB_Patient.address[0].postalCode

Außerdem instanziiert die Klasse *„PIOContent”* ein Objekt der Klasse „Path Validator“ (siehe Abb. 17). Während das „Root Object“ bei einem Import selbstständig XML-Pfade detektiert, die in der PIO-ULB Spezifikation nicht vorgesehen sind, übernimmt der „Path Validator“ diese Aufgabe bei einem Export. Sollte also ein Programmierfehler im Frontend vorliegen, sodass Daten unter einem Pfad abgelegt werden, der nicht spezifikationskonform ist, wird der Export fehlschlagen und die Generierung der XML-Datei abgebrochen. Eine entsprechende Fehlermeldung in der Konsole des Frontends stellt Detailinformationen zur Verfügung. Das Modul „Path Validator“ diente der Qualitätssicherung während der Entwicklungsphase und kann in Zukunft auf Änderungen in der PIO-ULB Spezifikation hinweisen.

Das Modul „Global Resources & Assets“ (siehe Abb. 17) beinhaltet neben globalen Konstanten, Typen und Interfaces auch ein PDF-Dokument, welches über den *„Disclaimer“*, der sich beim Starten des Editors automatisch öffnet, heruntergeladen werden kann. Das PDF-Dokument beinhaltet eine exakte Beschreibung des *PIO-Smalls*.

Zum Schluss listet Tabelle 5 alle Dateien auf, die im Quellcode des Backends enthalten sind, und ordnet diese einem Modul aus Abbildung 17 zu.

Tabelle 5: *Auflistung und Erklärung aller Dateien aus dem Quellcode des Backends*

Modul	Ordner / Datei	Beschreibung
Express-Server	express/server.ts	Starten des Express-Servers inkl. weiterer Initialisierungsschritte. Das start-Skript zum Starten des Backends (siehe package.json) führt unter anderem diese Datei aus.
	express/routes.ts	Definition aller API-Routen
	express/controllers.ts	Controller-Funktionen für alle API-Routen
Validator Service	ValidatorService.ts	Funktion, die Validierungsanfragen an den Validator Service weiterleitet.
Session Management	Session.ts	Für jeden angemeldeten Nutzer wird ein Session Objekt instanziiert, welches Nutzerdaten und das „Root-Object“ enthält.
	MultiSessionManager.ts	Beim Starten des Backends wird eine Instanz dieser Klasse erstellt. Sie verwaltet alle Sessions der angemeldeten Nutzer und schließt abgelaufene Sessions automatisch.
Address Book Handler & mongoDB	dataBase/AddressBookHandler.ts	Klasse mit Methoden für den Datenbankzugriff
	dataBase/dbConnection.ts	Funktionen zum Prüfen, Verbinden und Trennen der Datenbankverbindung. Vor jedem Datenbankzugriff durch den *AddressBookHandler* wird die Verbindung aufgebaut und nach jedem Zugriff wieder getrennt.
	dataBase/dbSchema.ts	Definiert die Struktur der mongoDB Datenbank.
	dataBase/exampleData.json	Enthält Beispieldaten, mit denen die Datenbank vorinitialisiert wird (nur in der *„webVersion“*).
	../docker-compose.yml	Docker-Datei zum Herunterladen und Ausführen des mongoDB Docker-Images in der Version 7.0.
Root Object	RootObject.ts	Klasse, die ein PIO-ULB repräsentiert.
	PIOHeader.ts	Klasse, die alle Header-Daten eines PIO-ULBs enthält.
	PIOContent.ts	Klasse, die alle inhaltlichen Daten eines PIO-ULBs enthält.
	generateCompositionSection.ts	Generiert alle Einträge der FHIR-Composition (= Sections).
Helper Functions	Helper.ts	Funktionen, die in mehreren Dateien des Backends Anwendung finden.
	RootObjectHelperToXml.ts	Funktionen die für die Serialisierungslogik (= toXML) benötigt werden.
	RootObjectHelperReadXml.ts	Funktionen die für die Deserialisierungslogik (= readXML) benötigt werden.
Path Validator	PathValidator.ts	Klasse zur Detektion invalider XML-Pfade bei der Generierung einer XML-Datei
Global Resources & Assets	@types/InterfacesForFHIRStructure.ts	Interfaces und Typen zur Abbildung der PIO-ULB Struktur
	@types/TypesForFHIRStructure.ts	
	@types/InterfacesForRootObject.ts	Interfaces und Typen, die im „Root Object“ verwendet werden
	@types/TypesForRootObject.ts	

	assets/SectionUuidTable.json	Definiert spezifikationskonform, welche FHIR-Ressourcen in welchem Composition-Eintrag (= Section) aufgelistet werden.
	GlobalVariables.ts	Globale Konstanten (z.B. Zeit der Inaktivität bis zur Schließung der Session)
	../assets/PIOEditorLimitationen-MitAnhang.pdf	PDF-Datei, die im *„Disclaimer“* des Editors heruntergeladen werden kann und eine genaue Beschreibung des *PIO-Smalls* beinhaltet.

6.4 Subsystem: Frontend-Server

Das Frontend ist ein eigenes node-Projekt, welches in *TypeScript* programmiert ist und die Frameworks *React* [12] und *Ant Design* [13] verwendet. *React* ist ein von *Meta Inc.* entwickeltes Framework, um benutzerfreundliche und interaktive User Interfaces (UI) für Webanwendung zu erstellen. Die Hauptaufgabe dieses Subsystems ist die nutzerfreundliche Visualisierung des PIO-ULB Standards sowie das Bereitstellen von Eingabemöglichkeiten zum Editieren eines PIO-ULBs. Alle HTTP-Anfragen des Frontends werden an das Backend geschickt. Da das Backend generisch implementiert wurde, müssen alle strukturbezogenen Informationen über die PIO-ULB Spezifikation im Frontend programmiert werden. Das Subsystem kann lokal durch das start-Skript in der *package.json* gestartet werden. Abbildung 20 zeigt schematisch die Bestandteile des Frontends:

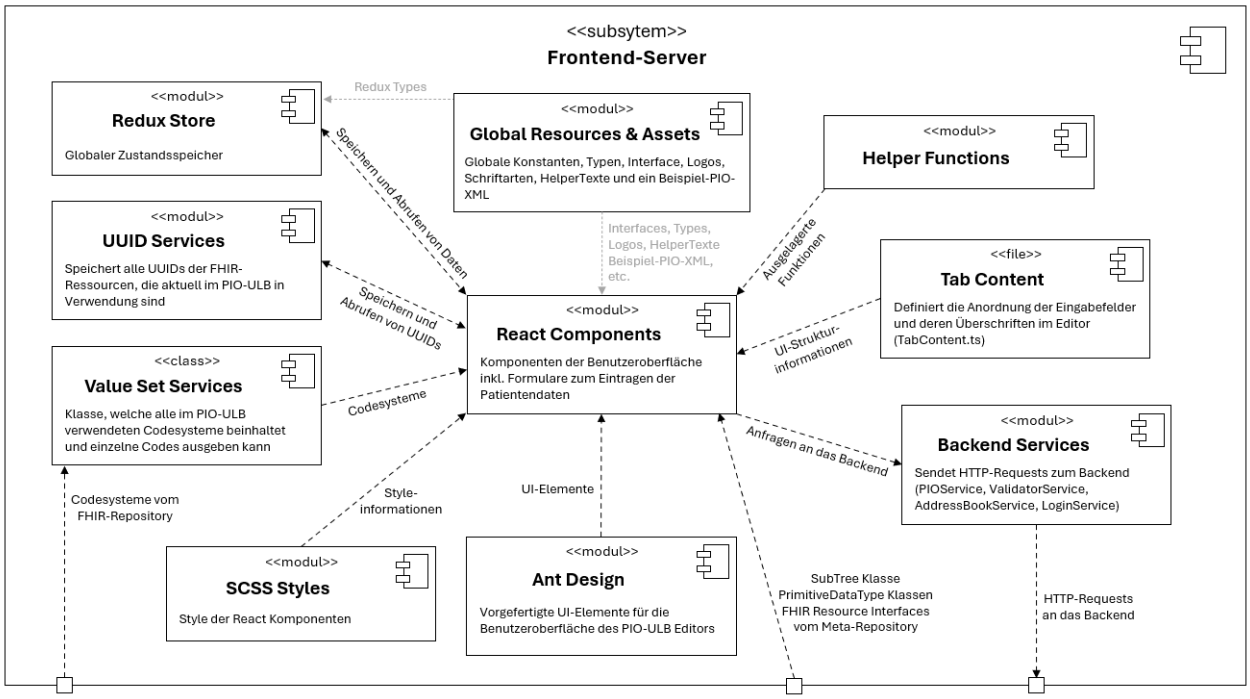

Abbildung 20: *Bestandteile des Subsystems „Frontend“ und deren Zusammenspiel*

Die „React Components“ bilden das zentrale Modul, welches die Struktur der Benutzeroberfläche unter Verwendung der „Ant Design“ UI-Komponenten definiert. Auf die

[12] Meta Open Source, *React – The libraryfor web and native user interfaces*, URL: https://react.dev/ [zuletzt aufgerufen: 11.03.2025].

[13] Ant Group and Ant Design Community, *Ant Design 5.0*, URL: https://ant.design/ [zuletzt aufgerufen: 11.03.2025].

Struktur der *React* Komponenten wird später eingegangen. Zuerst sollen alle anderen Module erläutert werden, beginnend mit den Services, die im Frontend bereitgestellt werden.

Die vier verschiedenen „Backend Services“ deklarieren Funktionen zur Kommunikation mit dem Backend. Jeder Service bündelt dabei thematisch verwandte Funktionen, die HTTP-Anfragen erstellen und absenden (siehe Tab. 6).

Tabelle 6: *Detaillierte Beschreibung der „Backend Services“*

Service	Beschreibung
LoginService	Dieser Service kümmert sich um das **Session Management**. Es können Sessions für neue Nutzer erstellt, geprüft und wieder geschlossen werden. Außerdem ist das Zurücksetzen der *Session Time* möglich, damit die Session erst dann automatisch geschlossen wird, wenn der Nutzer 4h bzw. 24h inaktiv war.
PIOService	Dieser Service stellt **PIO-ULB bezogene Funktionen** bereit. Es kann ein PIO-ULB geöffnet werden, indem ein XML-String gesendet und im Backend in ein „Root Object“ konvertiert wird. Weiterhin kann ein Export angestoßen und SubTrees gesendet bzw. empfangen werden. Dieser Service enthält alle Funktionen, um die Nutzereingaben an das Backend zu übermitteln und somit zu speichern.
AddressBookService	Dieser Service beinhaltet alle Funktionen für den lesenden und schreibenden Zugriff auf das Adressbuch, also die Datenbank im Backend.
ValidatorService	Dieser Service ermöglicht das senden eines XML-Strings, der dann vom Subsystem „Validator Service“ validiert wird.

Im Frontend werden zwei weitere Services verwendet, die jedoch nicht der Kommunikation mit dem Backend dienen. Der „Value Set Service“ ist eine Klasse, welche die Codesysteminformationen vom FHIR-Repository aufbereitet und für die React Komponenten bereitstellt. Der „UUID Service“ wird stehts mit den PIO-ULB Daten im Backend synchron gehalten und speichert alle verwendeten UUIDs der bereits erstellten FHIR-Ressourcen im Frontend. So können Nutzereingaben der richten Ressource zugeordnet werden.

Das Modul „Redux Store“ ist ein globaler Zustandsspeicher. Hier werden Anmeldedaten des Nutzers, Navigationsinformationen, die die Sichtbarkeit der einzelnen Screens steuern, die ID des aktuell ausgeführten Validierungsprozesses, Nutzereingaben, die über mehrere Eingabefelder hinweg synchron gehalten werden müssen, und die simulierten Adressbucheinträge in der *„webVersion“* (siehe Tab. 4) gespeichert.

Das Modul „Global Resources & Assets“ beinhaltet neben globalen Konstanten, Typen und Interfaces auch Schriftarten, Logos und Helper-Texte, die im Frontend angezeigt werden. Des Weiteren kann in der *„webVersion“* ein Demo-PIO-ULB zu Demonstrationszwecken geöffnet werden. Die XML-Demonstrationsdaten sind Teil dieses Moduls.

Das Modul „React Components“ beinhaltet alle Elemente, aus denen die Benutzeroberfläche besteht. Hier wird das Layout des Login-Screens, des Home-Screens, des Adressbuchs und der Editoransicht definiert. Das Design der einzelnen Elemente wird durch die „SCSS Styles“ vorgegeben. Die Editoransicht ermöglicht das Anzeigen und Editieren aller Daten in einem PIO-ULB und weist daher die komplexeste Struktur von allen Screens auf. Im Folgenden soll auf die Editoransicht im Detail eingegangen werden.

In der **Editoransicht** (siehe Abb. 5) werden dem Nutzer drei Reiter bzw. Tabs zur Verfügung gestellt. Das Modul „Tab Content“ definiert an einem zentralen Ort, den Inhalt und die Struktur der drei Tabs „Organisatorisches & Soziales“, „Pflegerische Infobausteine“ und „Aktuell nicht unterstützte Daten“. Für jeden Tab wird links ein einklappbares Seitenmenü angezeigt, welches automatisch mit den Zwischenüberschriften aus dem aktuell ausgewählten Tab befüllt wird.

Die ersten beiden Tabs beinhalten Eingabefelder, um den PIO-ULB zu befüllen. Diese Eingabefelder werden in Formularen der Ant-Design-Bibliothek thematisch gebündelt und repräsentieren ausschließlich das *PIO-Small*. Sowohl die inhaltliche Gliederung mit Hilfe der zwei Tabs als auch die Struktur innerhalb der Formulare orientiert sich stark an der PIO-ULB Spezifikation. Der dritte Tab (siehe Abb. 8) visualisiert Daten, die importiert wurden, nicht Teil des *PIO-Smalls* sind und somit keine eigenen Eingabefelder in den ersten beiden Tabs besitzen. Der dritte Tab ist nur bedingt sichtbar. Bevor genauer auf die Formulare eingegangen wird, soll anhand der Abbildung 21 die hierarchische Struktur der React Komponenten erklärt werden:

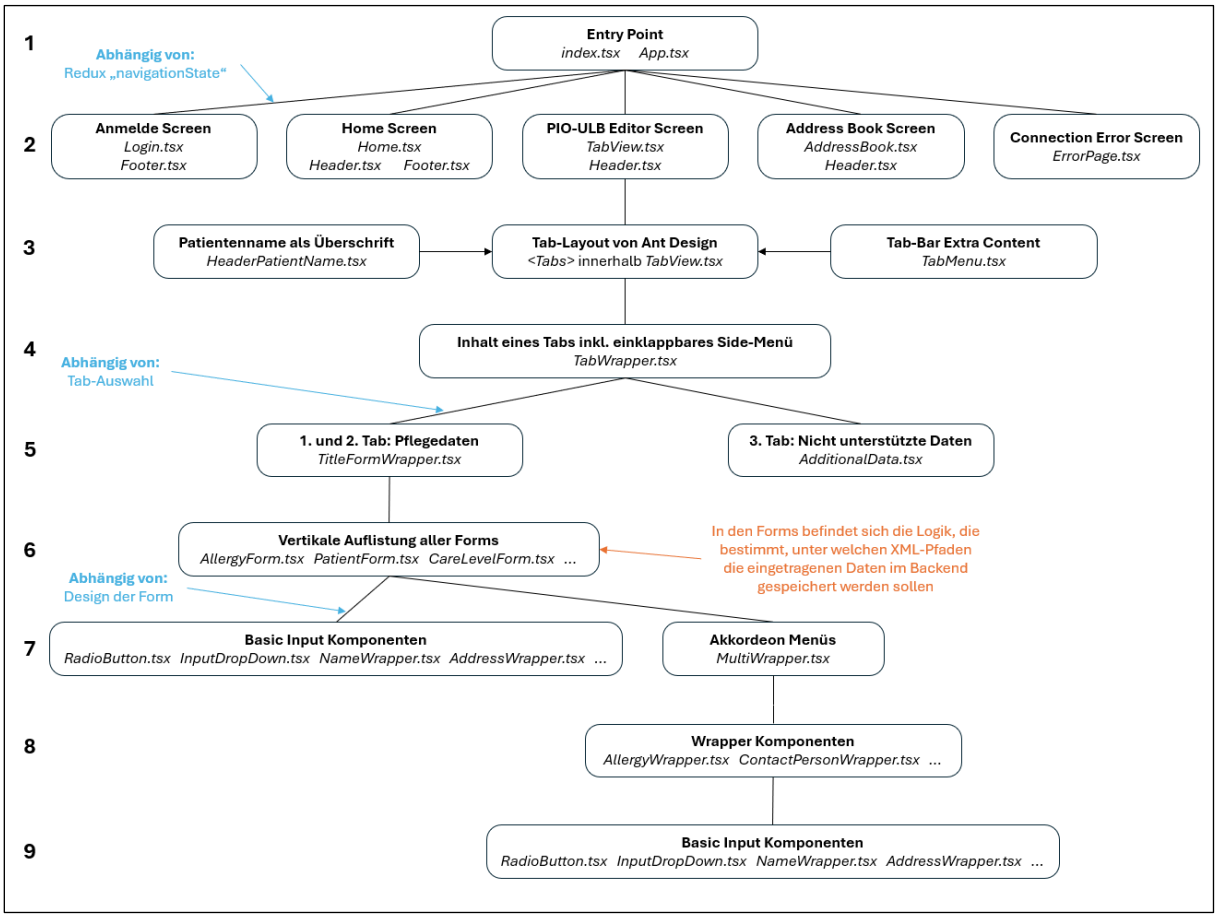

Abbildung 21: *Hierarchische Struktur der React Komponenten im Frontend inkl. Verweise auf spezielle Dateien*

Wie üblich bilden die Dateien *index.tsx* und *App.tsx* den Einstiegspunkt der React Applikation (siehe Abb. 21, Ebene 1). Je nach Navigationszustand der Applikation, welcher im Redux Store global gespeichert wird, sieht der Nutzer unterschiedliche Screens. Alle möglichen Screens sind auf Ebene 2 ersichtlich. Je nachdem welcher Screen aktuell angezeigt wird, werden *Header* und *Footer* ein- bzw. ausgeblendet. Hinter dem „PIO-ULB

Editor Screen“ verbirgt sich die oben beschriebene Editoransicht, auf die im Folgenden genauer eingegangen wird.

Die Editoransicht (siehe Abb. 5) besteht aus der *HeaderPatientName* Komponente, die den Patientennamen als große Überschrift im Screen darstellt, und einer <Tabs> Komponente, welche von der Ant-Design-Bibliothek bereitgestellt wird und die Tab-Funktionalität implementiert (Ebene 3). Die <Tabs> Komponente kann extra Inhalte rendern, die rechts oben neben der Tab-Bar angezeigt werden. Im Editor wird die *TabMenu* Komponente als extra Inhalt verwendet, welche den Validator-, Export- und Lösch-Button beinhaltet.

Unabhängig davon, welcher Tab aktiv ist, wird immer eine Wrapper-Komponente gerendert, die das einklappbare Seitenmenü enthält (Ebene 4). Das Seitenmenü ist also in jedem Tab sichtbar. Wenn der dritte Tab ausgewählt wird, ist die *AdditionalData* Komponente zusehen, während beim ersten und zweiten Tab eine weitere Wrapper-Komponente in der hierarchischen Struktur eingefügt wird, die sich um das Handling der Formulare kümmert (Ebene 5).

Je nachdem, welcher Tab aktiv ist, werden Formulare in einer vertikal scrollbaren Ansicht gerendert, die entweder zum Themengebiet „Organisatorisches & Soziales“ oder „Pflegerische Infobausteine“ gehören (Ebene 6). Auf die Funktionalität der Formulare wird später genauer eingegangen.

Ein **Formular** bündelt meistens alle Elemente einer FHIR-Ressource. Sieht die PIO-ULB Spezifikation vor, dass eine Ressource (z.B. Pflegegrad) nur einmal im gesamten PIO-ULB vorkommen darf, so greift das zugehörige Formular direkt auf Eingabefelder zurück, die als eigene *Basic Input* Komponenten implementiert wurden, um Wiederverwendbarkeit zu gewährleisten (Ebene 7, linker Zweig). Für Ressourcen, die öfter in einem PIO-ULB existieren können (z.B. Allergien), wurde eine *MultiWrapper* Komponente entwickelt, die denselben Inhalt mehrfach in einem Akkordeon Menü rendern kann (Ebene 7, rechter Zweig). Innerhalb des *MultiWrappers* werden weitere Wrapper-Komponenten angezeigt, die nur das User Interface eines Formulars ohne Logik bereitstellen (Ebene 8). Auf der letzten Hierarchieebene werden wieder die *Basic Input* Komponenten benutzt.

Die auf Ebene 8 eingesetzten Wrapper werden zum Beispiel auch in Modalen wiederverwendet. Modale sind Pop Up Fenster, die von Formularen aufgerufen werden, wenn eine zu referenzierende FHIR-Ressource noch nicht existiert, diese aber an Ort und Stelle vom Nutzer angelegt werden soll. Gibt der Nutzer zum Beispiel gerade die Daten einer medizinischen Diagnose ein, kann er auf einen Diagnosesteller referenzieren. Der Diagnosesteller ist meistens ein Arzt, der jedoch in einem anderen Formular angelegt wird. Damit der Nutzer in der Software nicht hin- und herspringen muss, falls der Arzt im PIO-ULB noch nicht existiert, gibt es die Funktion, die fehlenden Daten des Arztes an Ort und Stelle einzutragen. In diesem Fall öffnet sich ein Modal, welches die Daten des Arztes erfasst und speichert.

Da das Backend generisch implementiert wurde, muss das Frontend wissen, unter welchen XML-Pfaden die vom Nutzer eingetragenen Daten gespeichert werden sollen. Die Logik, welche die Daten an der richtigen Stelle in der XML-Struktur platziert, ist in den

Formularen zu finden. Die Formulare halten meist einen SubTree, der den für das Formular relevanten Ausschnitt aus der gesamten Datenstruktur enthält. Das Formular kann Daten im SubTree ändern und letztendlich zum Speichern an das Backend schicken.

Zum Schluss listet Tabelle 7 alle Dateien auf, die im Quellcode des Frontends enthalten sind, und ordnet diese einem Modul aus Abbildung 20 zu. Die React Komponenten werden in der Tabelle nicht nochmal aufgeführt, da sie schon in Abbildung 21 beschrieben wurden.

Tabelle 7: *Auflistung und Erklärung aller Dateien aus dem Quellcode des Frontends*

Modul	Ordner / Datei	Beschreibung
Backend-Services	services/AddressBookHandler.ts	Siehe Tabelle 6
	services/PIOService.ts	
	services/LoginService.ts	
	services/ValidatorService.ts	
UUID-Service	services/UUIDService.ts	Speichert alle bereits verwendeten UUIDs der FHIR-Ressourcen.
ValueSet-Service	services/ValueSetService.ts	Klasse, welche die Codesystem-Daten vom Meta-Repository für die Verwendung im Frontend aufbereitet.
Redux Store	redux/store.ts	Definiert die Struktur des Redux Stores und stellt diesen bereit.
	redux/initializeRedux.ts	Wenn ein PIO-ULB importiert wird, müssen gewisse Informationen aus dem Backend abgefragt werden, damit der Redux Store mit diesen initialisiert werden kann.
	redux/actions	Dieser Ordner enthält alle Redux Actions.
	redux/reducers	Dieser Ordner enthält alle Redux Reducers.
Global Resources & Assets	@types	Dieser Ordner enthält Typen und Interfaces, die für die Formulare, den Redux Store und die *Basic Input* Komponenten verwendet werden.
	assets/beispiel-pio.xml	Diese Datei enthält die Daten des Demo-PIO-ULBs.
	assets	Dieser Ordner enthält weitere Logos und Schriftarten.
	services/GlobalVariables.ts	Ein- und Ausschalten der Validator-Funktion
	services/HelperTextService.ts	Bereitstellung aller Helper-Texte, die bei manchen Eingabefeldern über das kleine „?“-Symbol abrufbar sind.
React Components & Tab Content	components	Dieser Ordner enthält alle React Komponenten, die schon in Abbildung 21 erklärt wurden.
	App.tsx	Entry Point für die React Applikation.
	index.tsx	
	components/ValidatorModal.tsx	React Komponenten, die über der gesamten Anwendung schweben und das Design sowie die Sichtbarkeit der Toast Messages und des Validator Pop Up Fensters steuern.
	components/ToastHandler.tsx	
SCSS Styles	styles	Dieser Ordner enthält alle scss-Dateien.

Helper Functions	services/HelperService.ts	Gewisse Helferfunktionen jeder Art.
	services/AuthorService.ts	Autoren-Handling, damit der Login-Name automatisch als Autor im PIO-ULB hinterlegt wird.
	services/DocumentService.ts	Helferfunktionen für die *UploadDocumentsForm* zum Hochladen von Bildern und PDF-Dateien.
	services/ReferenceService.ts	Helferfunktionen für das UUID Handling.
	services/SubTreeConverterService.ts	Manche Formulare speichern ihre Daten nicht in SubTrees, sondern in Form von Interfaces. Dies ist der Fall, wenn die Daten über mehrere Formulare synchron gehalten werden müssen und deshalb im Redux Store gespeichert sind. Diese Datei stellt Funktionen bereit, welche die Interfaces in SubTrees konvertieren und umgekehrt, damit sie vom Backend verarbeitet werden können.
	services/SubTreeHelperService.ts	Logik, die in mehreren Forms benötigt wird und deshalb ausgelagert wurde (z.B. zum Speichern einer Adresse).
	components/layout/Tabs/TabHelper.tsx	Funktion, die alle Einträge für das einklappbare Seitenmenü sammelt.
	services/PDFService.ts	Logik zum Exportieren eines PIO-ULBs in menschenlesbarer PDF-Form.

6.5 Subsystem: Validator Service

Der Validator Service ist ein eigenes .NET-Projekt, welches in *C-Sharp* programmiert ist. Da der PIO-ULB Editor nicht nur für das Erstellen von PIO-ULBs gedacht ist, sondern diese auch einlesen kann, ist die Validierung einer PIO-XML-Datei ein wichtiger Prozessschritt. Hierfür wurde der von „Firely“ [14] bereitgestellte Code verwendet, welcher FHIR-Strukturdefinitionen auswerten und anschließend gesamte FHIR-Bundles offline validieren kann. Der Ordner „WebAPI“ stellt den Entry Point dar, um den Validator Service zu starten. Abbildung 22 zeigt schematisch die Bestandteile des Validator Services:

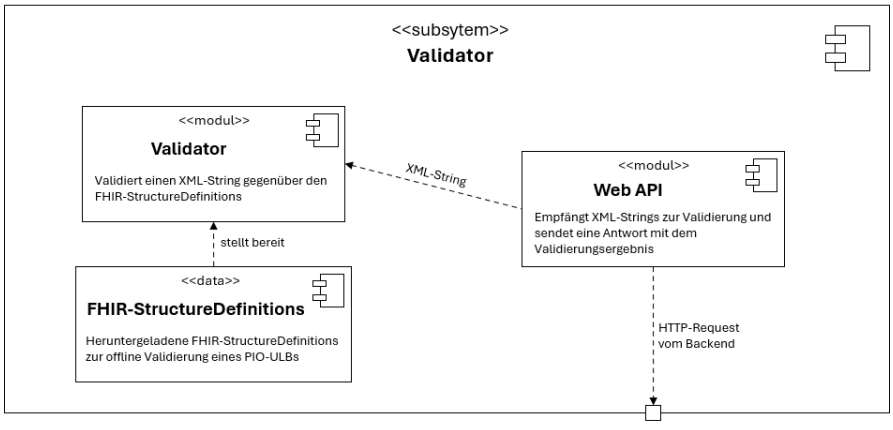

Abbildung 22: *Bestandteile des Subsystems „Validator Service“ und deren Zusammenspiel*

Der XML-String, der validiert werden soll, wird vom Frontend über das Backend zum Validator Service mittels HTTP-Anfrage geschickt. Das „Web API“ Modul empfängt die Anfrage und leitet den XML-String zum „Validator“ Modul weiter, welches ungefähr 30 Sekunden

[14] Firely .NET SDK, *Validation of FHIR data*, URL: https://docs.fire.ly/projects/Firely-NET-SDK/en/latest/validation.html [zuletzt aufgerufen: 11.03.2025].

für einen Validierungsprozess benötigt. Das Ergebnis wird wieder zum Frontend zurückgeschickt und dort visualisiert. Damit die einzelnen Validierungsprozesse im Frontend richtig zugeordnet werden können, wird für jeden gestarteten Prozess eine ID vergeben. Im Redux Store wird die ID des aktuellsten Prozesses gespeichert. Dies ist nötig, da der Nutzer innerhalb der 30 Sekunden zwei Validierungsprozesse anstoßen könnte. Die FHIR-Strukturdefinitionen, auf Basis dessen die Validierung durchgeführt wird, sind im Subsystem hinterlegt. Weiterhin musste der „Firely" Validator entsprechend der *PIO-Small* Spezifikation angepasst werden (Herausfiltern von Fehlermeldungen, die aufgrund der Umfangsreduzierung auftreten).

Zum Schluss listet Tabelle 8 alle Dateien auf, die im Quellcode des Validator Services enthalten sind, und ordnet diese einem Modul aus Abbildung 22 zu.

Tabelle 8: *Auflistung und Erklärung aller Dateien aus dem Quellcode des Validator Services*

Modul	Ordner / Datei	Beschreibung
Web API	src/WebAPI/Program.cs	Bereitstellung einer API-Schnittstelle
	src/WebAPI/Response.cs	Klasse, die die Struktur der Antwort definiert.
Validator	src/Firely.Fhir.Validation.R4/ PIOEditorValidator.cs	Hier findet der Validierungsprozess mit „Firely" statt. Außerdem werden Fehlermeldungen, die aufgrund der Reduzierung auf das *PIO-Small* entstehen, herausgefiltert.
FHIR-Structure-Definitions	src/Firely.Fhir.Validation.R4/ structureDefinitions	Dieser Ordner enthält alle FHIR-Strukturdefinitionen, die für den Validierungsprozess benötigt werden.

7. Laufzeitsicht

In der Laufzeitschicht sollen dynamische Szenarien des PIO-ULB Editors anhand von UML-Sequenzdiagrammen erläutert werden. Dabei wird hauptsächlich die Kommunikation zwischen Front- und Backend betrachtet, in speziellen Fällen werden auch die dynamischen Abläufe im Backend genauer beschrieben. In diesem Kapitel soll auf folgende dynamische Szenarien eingegangen werden:

- Login eines Nutzers
- Logout eines Nutzers
- Neuen PIO-ULB anlegen
- Dateneingabe in einen PIO-ULB
- Anlegen eines neuen Adressbuch-Eintrags
- Export eines PIO-ULBs im XML- bzw. PDF-Format / Serialisierungslogik (inkl. Abläufe im Backend)
- Import eines PIO-ULBs im XML-Format / Deserialisierungslogik (inkl. Abläufe im Backend)

Alle abgebildeten Sequenzdiagramme visualisieren ausschließlich Szenarien der *„local-Version“* (siehe Tab. 4). Auf die *„webVersion“* wird (wenn nötig) im Text eingegangen. Die Kommunikation zwischen Front- und Backend läuft über die Backend Services (im Frontend) und den Express-Server (im Backend). Diese Module werden jedoch aus Gründen der Übersichtlichkeit in den Sequenzdiagrammen nicht abgebildet.

7.1 Login eines Nutzers

Das dynamische Szenario „Login eines Nutzers“ beschreibt den Prozess, in dem ein Nutzer seinen Vor- und Nachnamen im Login-Screen des PIO-ULB Editors einträgt, auf den Anmelde-Button klickt und bei einem erfolgreichen Login zum Home-Screen weitergeleitet wird. Abbildung 23 stellt das entsprechende Sequenzdiagramm dar:

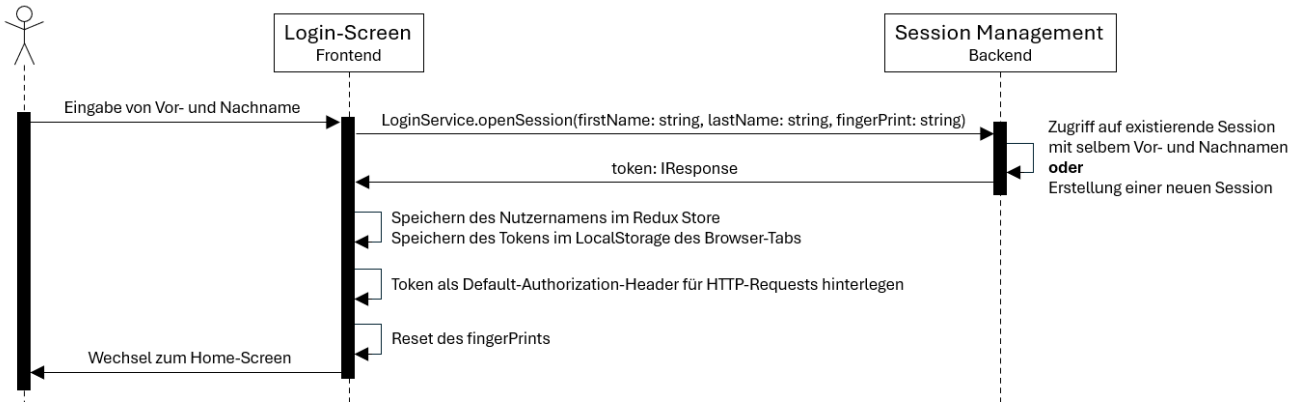

Abbildung 23: *Szenario „Login eines Nutzers“ als UML-Sequenzdiagramm*

Bei einem erfolgreichen Login wird im Backend ein JSON Web Token (JWT) generiert, welcher an das Frontend übermittelt wird. Das Frontend speichert den Token und schickt ihn

bei jeder weiteren HTTP-Anfrage mit, sodass die richtige Session identifiziert werden kann. Der Token besitzt eine zeitlich begrenzte Gültigkeit (siehe Tab. 4).

Um Logins mit demselben Vor- und Nachnamen zu ermöglichen, wird eine einzigartige ID („Fingerprint“), für jeden Frontend-Client neu generiert. Der „Fingerprint“ findet ausschließlich in der *„webVersion“* Anwendung.

7.2 Logout eines Nutzers

Das Szenario „Logout eines Nutzers“ stellt den komplementären Prozess zum Login dar. Wenn der Nutzer im Kopfzeilenmenü des PIO-ULB Editors auf das Nutzer-Icon klickt, öffnet sich ein Popover, in dem ein Logout-Button zu finden ist. Dieser Button stößt den Logout-Prozess an (siehe Abb. 24).

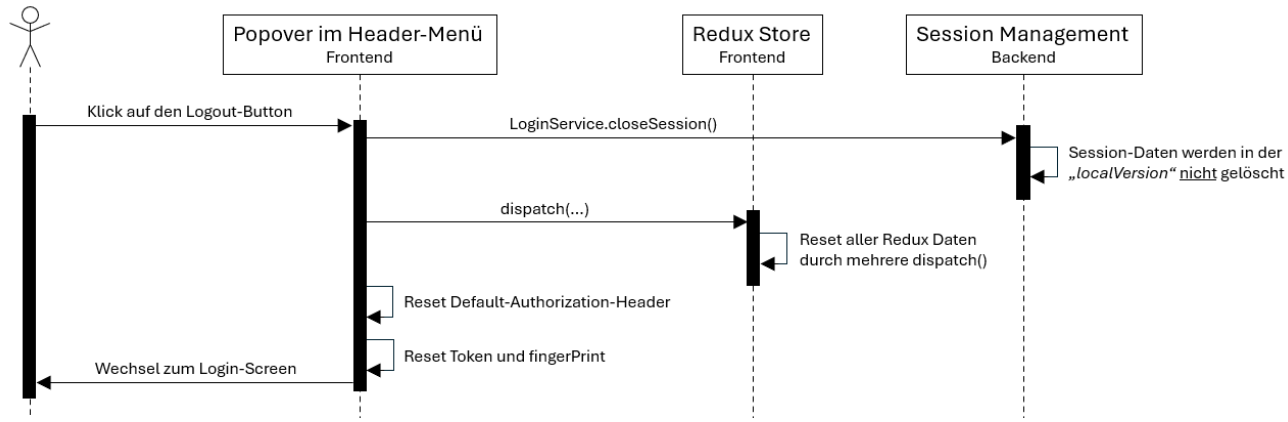

Abbildung 24: *Szenario „Logout eines Nutzers“ als UML-Sequenzdiagramm*

In der *„localVersion“* wird die Session im Backend bei einem Logout (oder bei einem Schließen des Browser-Tabs) nicht gelöscht. Durch einen erneuten Login mit dem selben Namen kann wieder auf die Session zugegriffen werden. Nur in der *„webVersion“* werden die Session-Daten unwiderruflich gelöscht.

Alle weiteren Logout-Prozesse, die in Abbildung 24 zu sehen sind, wurden im Programmcode in einer Funktion als Teil des LoginServices gebündelt. Außerdem ist zu erkennen, dass eine *dispatch*-Anfrage an den Redux Store keine Antwort erwartet, sondern asynchron zu anderen Prozessen abgearbeitet werden kann.

7.3 Neuen PIO-ULB anlegen

Dieses Szenario beschreibt den Prozess, in dem der Nutzer beim Home-Screen auf „PIO-ULB erstellen“ klickt und daraufhin die Editoransicht mit leeren Eingabefeldern angezeigt wird. Abbildung 25 zeigt alle Aktivitäten, die von der Software im Hintergrund abgearbeitet werden.

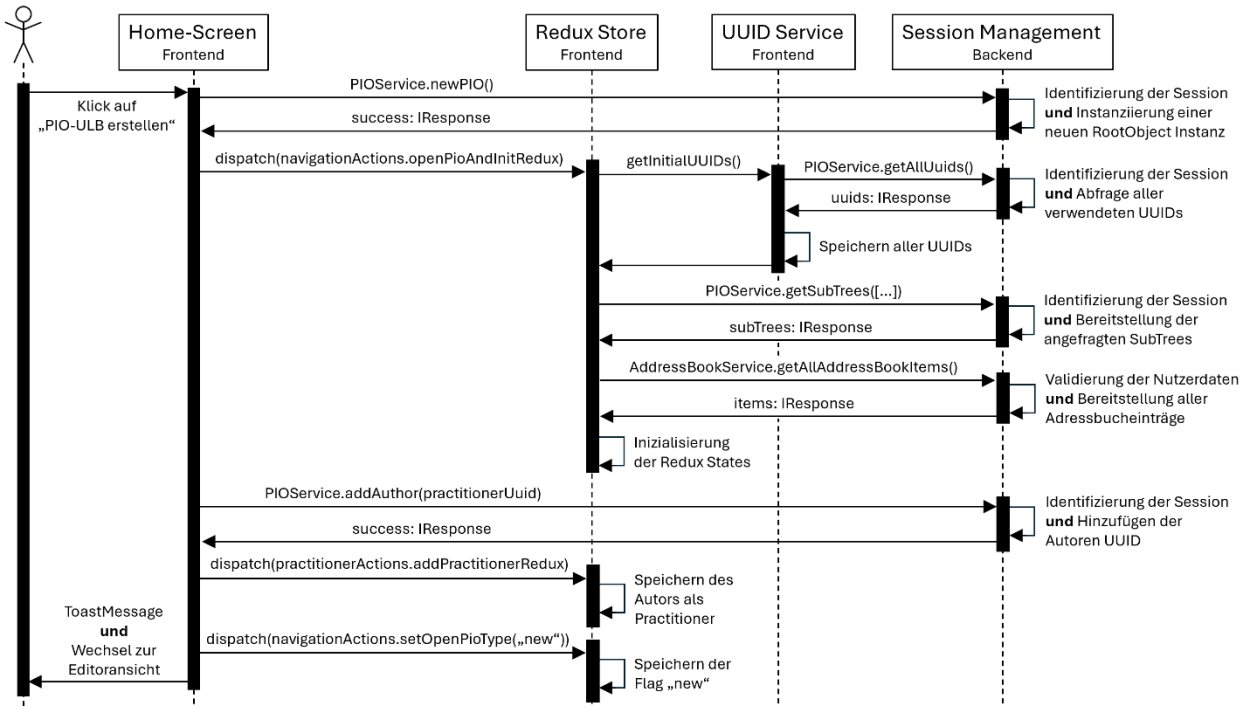

Abbildung 25: *Szenario „Neuen PIO-ULB anlegen“ als UML-Sequenzdiagramm*

Nachdem der Nutzer auf den Button geklickt hat, wird im Backend ein neues RootObject für die entsprechende Session erstellt. Sollte bereits ein RootObject für die Session existieren, wird der Nutzer darüber informiert. In diesem Fall wird erst nach einer aktiven Zustimmung des Nutzers das alte RootObject überschrieben. Dieser Sonderfall wurde im Programmcode berücksichtigt, im Sequenzdiagramm aber nicht abgebildet.

Wenn das Backend erfolgreich ein neues RootObject erstellt hat, wird eine Redux-Action ausgelöst. Diese Action kümmert sich sowohl um das Aktualisieren aller UUIDs, die im UUID Service gespeichert sind, als auch um die Abfrage von allen Adressbuchenträgen und gewissen vordefinierten SubTrees aus dem Backened, um die entsprechenden Redux States zu initialisieren. Der Prozess, der durch die Funktion *getInitialUUIDs()* angestoßen wird, frägt die UUIDs von allen bereits erstellten Ressourcen im Backend ab und speichert sie im UUID Service. Dieser Prozess ist für das Anlegen eines neuen PIO-ULBs irrelevant, da im Backend noch keine Ressourcen erstellt wurden. Erst beim Import einer XML-Datei (siehe Kapitel 7.7) ist diese Funktion von Relevanz. Um redundanten Code zu vermeiden, wird beim Anlegen eines neuen PIO-ULBs und beim Import einer XML-Datei, dieselbe Redux-Action verwendet.

Außerdem wird automatisch eine FHIR-Practitioner-Ressource angelegt. Practitioner Ressourcen bilden behandelnde Personen ab, in diesem Fall also den Autor des PIO-ULBs, weil dies vom PIO-ULB Standard so gefordert wird. Der Vor- und Nachname, welcher vom Nutzer während des Logins eingetragen wurde, kann nun automatisch in die Practitioner-Ressource übertragen werden.

Weiterhin wird mit der Flag *„new“* im Redux Store festgehalten, dass ein PIO-ULB neu angelegt wurde wurde. Dies wirkt sich auf das Validierungsverhalten aus, weil ausschließlich beim Import die Validierung angestoßen werden soll. Das Anlegen eines neuen PIO-

ULBs initiiert keine Validierung, weshalb der entsprechende Button im Menü der Tab-Bar fehlt.

Zum Schluss wird dem Nutzer eine Nachricht über das erfolgreiche Anlegen des neuen PIO-ULBs angezeigt. Der Screen wechselt daraufhin zur Editoransicht, in der der Autor bereits als behandelnde Person hinterlegt ist.

7.4 Dateneingabe in einen PIO-ULB

In diesem Szenario befindet sich der Nutzer in der Editoransicht und gibt Daten in ein Eingabefeld ein. Anschließend klickt er aus dem Eingabefeld heraus. Durch das „Herausklicken“ wir der Prozess zum Speichern der Daten angestoßen, welcher in Abbildung 26 dargestellt ist. Einige wenige Eingabefelder validieren die Nutzereingabe bevor die Daten an das Backend geschickt werden. Dies ist in Abbildung 26 nicht dargestellt.

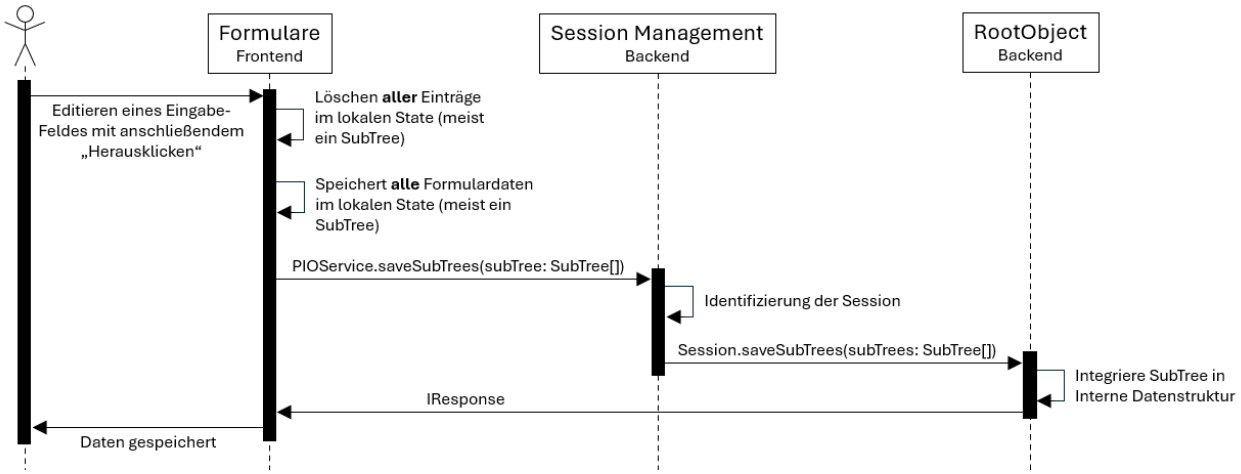

Abbildung 26: *Szenario „Dateneingabe in einen PIO-ULB“ als UML-Sequenzdiagramm*

Wie in Kapitel 6.4 bereits beschrieben, speichert jedes Formular die eingegebenen Daten in einem lokalen State. Dieser lokale State ist meist ein SubTree, welcher den relevanten Ausschnitt aus der Gesamtstruktur des PIO-ULBs darstellt. Wenn vom Nutzer neue Daten eingegeben werden, löscht das Formular alle Daten aus dem lokalen State, um anschließend alle bereits eingegebenen Daten wieder in den State zu schreiben. Durch dieses Vorgehen wurde der Programmieraufwand reduziert, da nicht jedes Hinzufügen oder Herauslöschen von Daten in einem Eingabefeld separat behandelt werden muss.

Falls im lokalen State kein SubTree gehalten wird, muss dieser in einen SubTree konvertiert werden, bevor er an das Backend geschickt wird. Dieser seltene Konvertierungsschritt ist in Abbildung 26 nicht abgebildet.

Sobald der beschriebene SubTree im Frontend vorliegt, wird dieser an das Backend geschickt. Dort wird der SubTree in das zugehörige RootObject eingefügt. Nun sind alle Daten des Formulars im Backend gespeichert. Der Nutzer erhält keine visuelle Benachrichtigung über die erfolgreiche Speicherung.

7.5 Anlegen eines neuen Adressbuch-Eintrags

Dieses Szenario beschreibt das Anlegen eines neuen Adressbuch-Eintrags über den Adressbuch-Screen, welcher über das Kopfzeilenmenü aufgerufen werden kann. Das Adressbuch kann Informationen über Organisationen (z.B. Pflegeheim oder Krankenhaus) speichern. Das Sequenzdiagramm in Abbildung 27 visualisiert alle Programmabläufe, die vom Klick auf den Button „Neuen Eintrag hinzufügen“ bis zum Erscheinen des neuen Adressbuch-Eintrags softwareintern stattfinden.

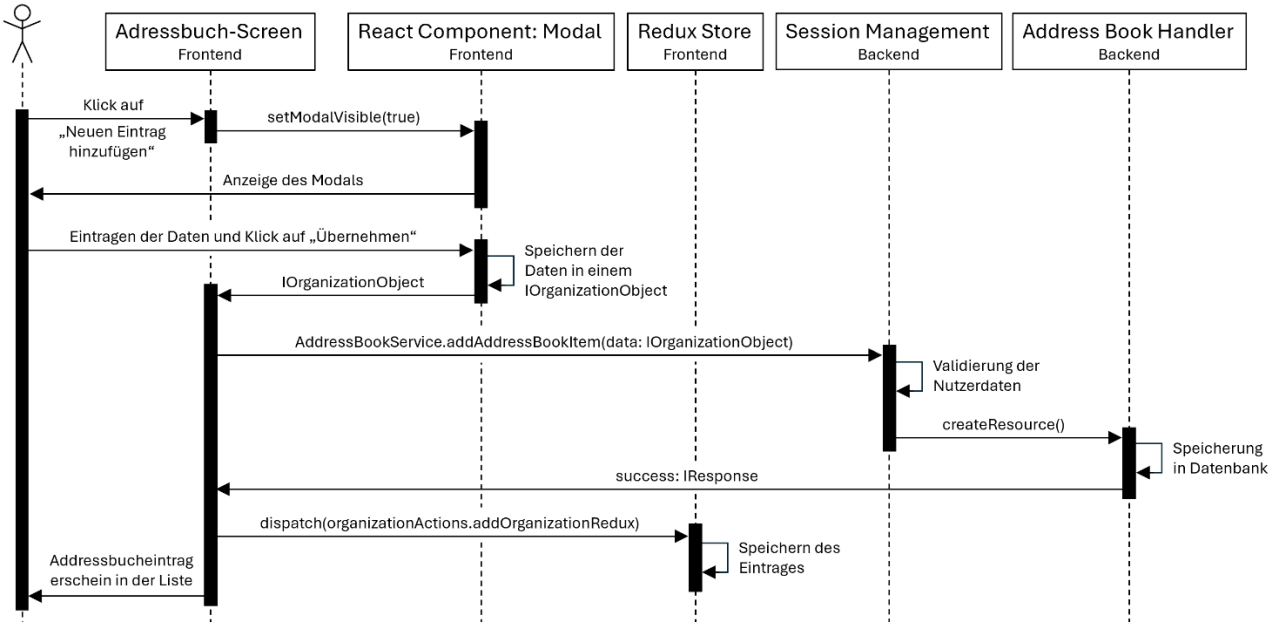

Abbildung 27: *Szenario „Anlegen eines neuen Adressbuch-Eintrags“ als UML-Sequenzdiagramm*

Das Modal ist eine React Komponente, welches sich wie ein Popup Fenster über dem aktiven Screen öffnet und eine Dateneingabe für verschiedene Kontexte ermöglicht. In diesem Fall wird der Inhalt der *OrganizationForm* im Modal gerendert, sodass der Nutzer Daten einer Organisation eintragen kann. Wenn der Nutzer auf „Übernehmen“ klickt, bündelt das Modal alle eingetragenen Daten in einem *IOrganizationObject*. Dieses Interface wird vom Meta-Repository bereitgestellt und bildet alle Informationen einer FHIR-Organisationsressource in flacherer Hierarchie ab. Das *IOrganizationObject* wird anschließend über den *AddressBookService* an das Backend geschickt. Sobald das Session Management die Nutzerdaten erfolgreich validiert hat, können die Organisationsdaten in die Datenbank geschrieben werden. Weiterhin werden die Daten im Redux Store gespeichert. Alle Adressbuch-Einträge aus der Datenbank werden mit dem Redux Store stets synchron gehalten, damit Eingabefelder, in denen alle Organisationen aus dem Adressbuch selektierbar sein sollen, aus dem Redux Store vorinitialisiert werden können.

Sollte die Validierung der Nutzerdaten fehlschlagen, ist der Session-Token abgelaufen und der Nutzer wird zum Login-Screen weitergeleitet. Eine Popup-Nachricht informiert den Nutzer über die abgelaufene Session. Dieser Fehlerfall ist im Sequenzdiagramm nicht abgebildet.

Außerdem gibt es zwei weitere Fälle, in denen neue Adressbuch-Einträge angelegt werden. Diese Fälle sind im Sequenzdiagramm nicht zu sehen, werden aber im Folgenden kurz beschrieben:

In der Editoransicht existieren Dropdown-Eingabefelder, in denen Organisationen ausgewählt werden können. Sollte die gewünschte Organisation fehlen, kann der Nutzer eine Organisation zum Adressbuch an Ort und Stelle hinzufügen, indem er auf „Neue Einrichtung hinzufügen“ klickt. Daraufhin öffnet sich das Modal, welches bereits weiter oben erklärt wurde.

Der zweite Fall, in dem Adressbuch-Einträge angelegt werden, stellt der Import eines PIO-ULBs dar. Diese Einträge werden automatisch angelegt, ohne das eine Nutzereingabe erforderlich ist. Enthält der PIO-ULB Organisationen, welche im Adressbuch noch nicht vorhanden sind, werden diese automatisch hinzugefügt. Die Identifizierung, ob Organisationen aus dem PIO-ULB bereits im Adressbuch vorhanden sind, erfolgt über die UUID der Organisationsressourcen (siehe Kapitel 12). Das automatische Hinzufügen von Organisationen zum Adressbuch ohne Nutzerzustimmung ist keine optimale Lösung und sollte in Zukunft nochmal überdacht werden.

7.6 Export eines PIO-ULBs im XML- bzw. PDF-Format / Serialisierungslogik

In diesem Abschnitt wird der Export einer XML- bzw. PDF-Datei näher betrachtet. Wenn sich der Benutzer in der Editoransicht befindet, hat er über den Export-Button im Menü der Tab-Bar die Möglichkeit, den PIO-ULB als XML- oder PDF-Datei herunterzuladen. Der Downloadprozess ist beendet, sobald die Datei über den Downloadmanager des verwendeten Browsers heruntergeladen wurde. Abbildung 28 zeigt, welche softwareinternen Prozesse bis zu einem erfolgreichen Download stattfinden:

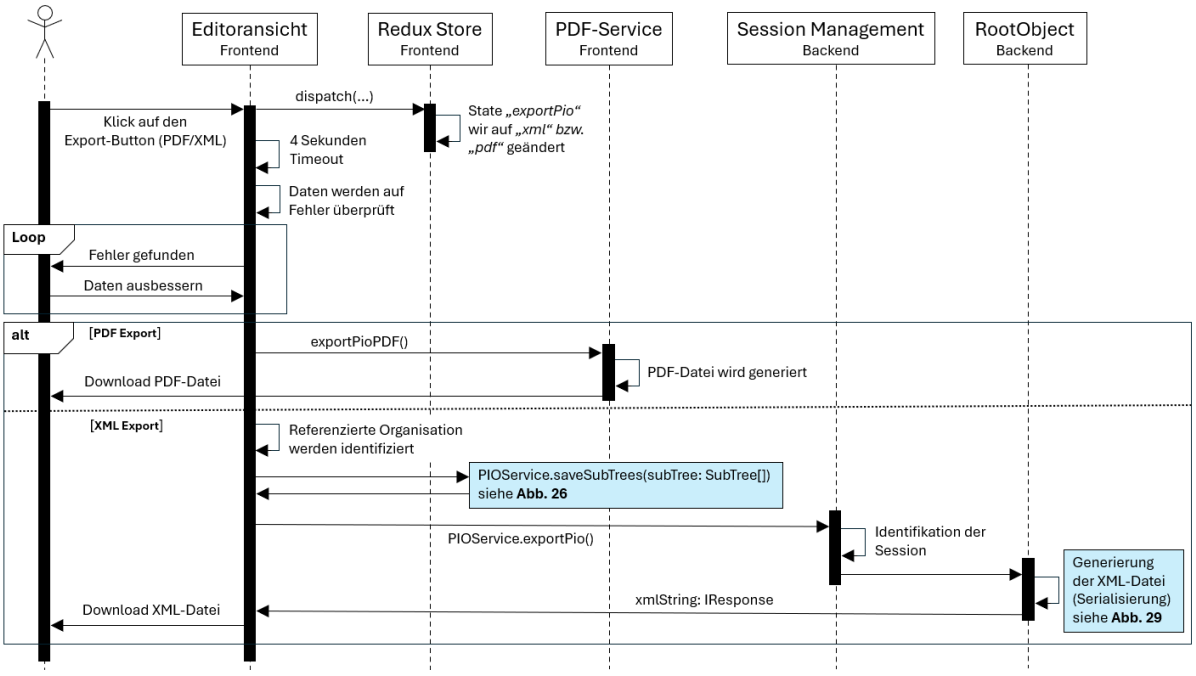

Abbildung 28: *Szenario „Export eines PIO-ULBs im XML- bzw. PDF-Format“ als UML-Sequenzdiagramm*

Zunächst wir der State *„exportPio“* im Redux Store auf *„xml“* oder *„pdf“* gesetzt, je nach gewählter Exportoption. Sobald dieser State nicht mehr leer ist, sondern einen String-Wert beinhaltet, werden automatisch alle Akkordeonmenüs in der Editoransicht aufgeklappt. Das Aufklappen der Menüs dauert eine Weile, weshalb die Software mit einem Timeout von 4 Sekunden auf die Vollendung des Aufklappvorgangs wartet. Unmittelbar nachdem alle Akkordeonmenüs aufgeklappt und somit alle Eingabefelder sichtbar sind, wird eine Frontendvalidierung aller Eingabefelder angestoßen. Diese Validierung wird unabhängig von der gewählten Exportoption (XML oder PDF) gestartet und fokussiert sich ausschließlich auf die Überprüfung von Pflichtfeldern. Sollte ein Pflichtfeld nicht ausgefüllt sein, wird der Nutzer über den Fehler informiert und der sichtbare Screen springt automatisch zum betroffenen Eingabefeld. Iterativ wird der Nutzer über alle weiteren leeren Pflichtfelder informiert, bis alle Fehler behoben wurden.

Wird die Validierung erfolgreich abgeschlossen, kann im Exportprozess der nächste Schritt eingeleitet werden. Abhängig von der gewählten Exportoption (XML oder PDF) werden unterschiedliche Prozesse durchgeführt:

Der **PDF-Export** erfolgt vollständig im Frontend. Die Funktion *exportPioPDF()* ruft den PDF-Service auf, der einen digitalen Screenshot der Daten erstellt. Hierzu werden die Informationen als HTML-Elemente gespeichert und in ein PDF-Dokument umgewandelt. Zusätzlich wird das Styling für eine bessere Lesbarkeit angepasst. Das PDF-Dokument enthält zusätzlich ein Signaturfeld und hochgeladene Anhänge wie Arztbriefe oder Atteste werden automatisch der PDF-Datei hinzugefügt. Nach der Generierung wird die PDF-Datei dem Benutzer direkt über den Downloadmanager des Browsers bereitgestellt.

Beim **XML-Export** müssen zunächst referenzierte Organisationen identifiziert und in den PIO-ULB eingefügt werden, da sie sich bisher nur im Adressbuch befanden. Damit die Organisationsdaten korrekt übernommen werden, müssen die Daten aus dem Adressbuch in SubTrees umgewandelt und an das Backend geschickt werden (siehe Abb. 28). Sobald die Organisationsdaten im Backend gespeichert wurden, kann der XML-Export im Backend gestartet werden.

Abbildung 29 beschreibt detailliert, wie ein RootObject in einen XML-String konvertiert wird (**Serialisierungslogik**). Hierbei werden nur Prozesse im Backend betrachtet. Wenn der Nutzer ein PIO-ULB als XML-Datei exportieren möchte, wird die Methode *„toXML()“* des RootObjects aufruft. Mit dem Aufruf der Methode startet der Prozess, der in Abbildung 29 dargestellt ist.

Damit ein valides PIO-ULB erstellt werden kann, muss sowohl mindestens ein Autor in den Header-Daten gesetzt sein als auch genau eine FHIR-Patienten Ressource vorhanden sein, weil sich ein PIO-ULB immer auf genau einen Patienten bezieht. Ist kein Autor eingetragen oder existiert keine bzw. mehr als eine FHIR-Patienten Ressource, wird ein Fehler ausgegeben und der Prozess abgebrochen.

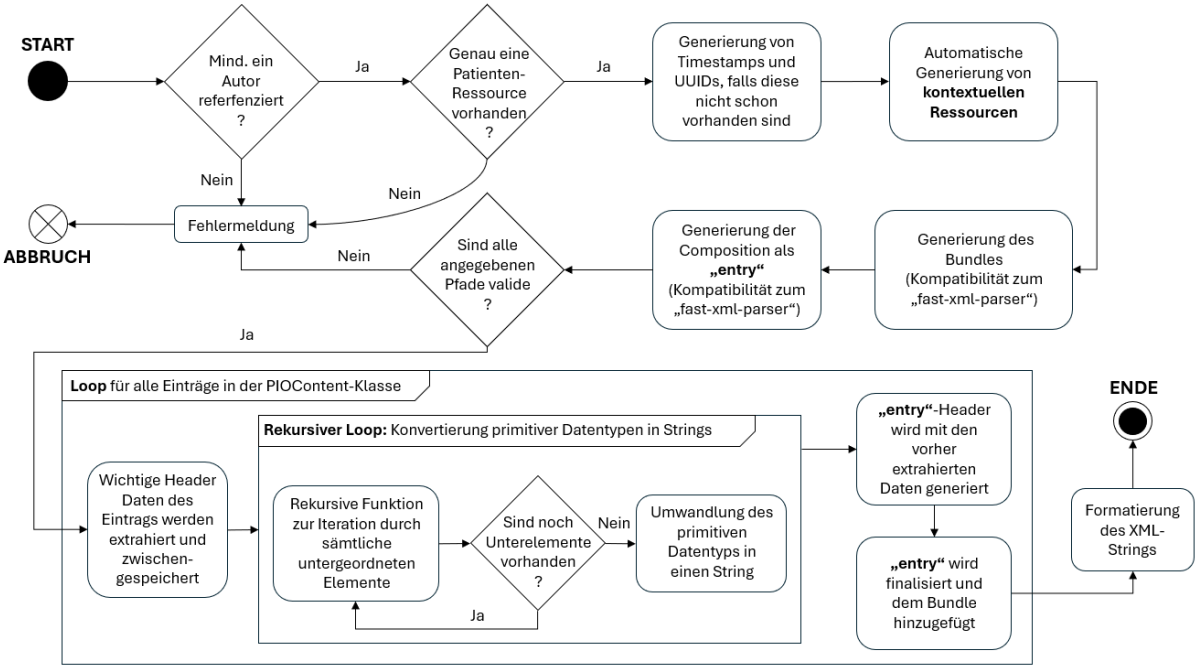

Abbildung 29: *Exportprozess im Backend als UML-Aktivitätsdiagramm (Serialisierungslogik)*

Anschließend werden Zeitstempel und UUIDs generiert, die im Bundle und in der Composition als Header-Daten verwendet werden. Falls der PIO-ULB importiert wurde und daher bereits UUIDs für das Bundle und die Composition vorhanden sind, werden natürlich die bereits existierenden UUIDs wiederverwendet. Die Zeitstempel werden jedoch aktualisiert.

Im nächsten Schritt werden sogenannte **kontextuelle Ressourcen** erzeugt, die nicht von expliziten Nutzereingaben abhängen und daher automatisch generiert werden können. Es gibt genau acht solcher Ressourcen, die dem RootObject automatisch hinzugefügt werden, um die Vollständigkeit der XML-Datei und dessen Konformität zum PIO-ULB Standard zu gewährleisten. Diese Ressourcen geben lediglich das Vorhandensein von anderen Ressourcen an und referenzieren diese. Beispielsweise werden in der Ressource *„PresenceAllergies“* alle vorhandenen Allergie-Ressourcen (also alle Allergien des Patienten) referenziert und somit gebündelt. Weißt der Patient Allergien auf, wird in der Ressource *„PresenceAllergies“* der Wert *„present“* (= Vorhanden) hinterlegt. Sollte der Patient keine Allergien haben, wird der Wert *„absent“* (= Abwesend) gespeichert. Die PIO-ULB Spezifikation sieht vor, dass ebenfalls ein Wert *„unknown“* (= unbekannt) hinterlegt werden kann, falls das Vorhandensein von Allergien unbekannt ist. Aus einer pflegerischen Perspektive ist es ein Unterschied, ob der Patient keine Allergien besitzt oder Allergieinformationen nicht vorliegen, also unbekannt sind. Im Rahmen der *PIO-Small* Reduktion wurde jedoch der Wert *„unknown“* aus allen kontextuellen Ressourcen herausgekürzt, also im PIO-ULB Editor nicht umgesetzt.

Anschließend wird das Bundle erstellt, welches wichtige Header-Daten (UUIDs, Zeitstempel, usw.) enthält. Alle weiteren Daten werden als *„entry“* im Bundle hinterlegt. Jeder *„entry“* repräsentiert eine eigene FHIR-Ressource, wobei die Composition (=

Inhaltsverzeichnis) auch eine eigene Ressource darstellt. Bei der Generierung des Bundles und aller *„entries“* wird stets auf die Kompatibilität zum verwendeten „fast-xml-parser“ [15] geachtet.

Vor der Verarbeitung aller weiteren *„entries“*, welche in der PIOContent Klasse als „EntryType“ (siehe Abb. 19) gespeichert sind, wird mit Hilfe des Moduls „PathValidator“ (siehe Abb. 17) überprüft, ob alle beschriebenen Pfade valide sind. Falls ungültige Pfade vorhanden sind, wird ein Fehler ausgegeben und der Prozess abgebrochen. Sind alle Pfade gültig, wird jeder Eintrag der PIOContent Klasse iterativ durchlaufen, um für jeden Eintrag in der PIOContent Klasse ein *„entry“* im Bundle zu erzeugen.

Für jede Iteration werden zuerst die relevanten Header-Daten extrahiert und zwischengespeichert. Anschließend erfolgt eine rekursive Umwandlung der primitiven Datentypen (siehe Kapitel 9.1) in Strings. Hierbei wird an jedem Knotenpunkt überprüft, ob ein primitiver Datentyp vorhanden ist, und die rekursive Funktion für alle weiteren Unterelemente aufgerufen. Alle detektierten primitiven Datentypen werden in einen String umgewandelt, sodass der „fast-xml-parser“ die Daten korrekt verarbeiten kann. Nach der Umwandlung der Datentypen wird der *„entry“*-Header mit den zuvor extrahierten Daten erstellt. Der erstellte *„entry“* wird schließlich dem Bundle hinzugefügt.

Zum Schluss wird der XML-String formatiert und an das Frontend zurückgeschickt. Die XML-Datei wird im Frontend automatisch über den Downloadmanager des verwendeten Browsers heruntergeladen (siehe Abb. 28).

7.7 Import eines PIO-ULBs im XML-Format / Deserialisierungslogik

Dieses Szenario beschreibt alle Hintergrundprozesse, die während dem Import einer XML-Datei ablaufen und für den Nutzer nicht direkt ersichtlich sind. Das Szenario startet mit einem Klick auf „PIO-ULB importieren“ im Home-Screen und der anschließenden Auswahl einer XML-Datei aus dem Dateiexplorer des Betriebssystems. Daraufhin werden alle Daten aus der XML-Datei ausgelesen und in die Eingabefelder der Editoransicht übertragen. Dies dauert je nach Umfang der XML-Datei ungefähr zwei Sekunden. Der Importprozess ist vollkommen abgeschlossen, sobald das Validierungsergebnis vom Validator Service vorliegt, was bis zu 30 Sekunden dauern kann. Abbildung 30 zeigt das zugehörige Sequenzdiagramm.

Dem Nutzer ist es nur möglich, Textdateien zu öffnen. Andere Dateiformate wie Bilder oder PDF-Dokumente werden dem Nutzer im Dateiexplorer erst gar nicht angezeigt. Nachdem der Nutzer eine Textdatei zum Importieren ausgewählt hat, findet eine Inputvalidierung im Frontend statt, die lediglich prüft, ob die Datei leer ist. Anschließend wird der String, der aus der Textdatei ausgelesen wurde, an das Backend geschickt. Die Umwandlung des XML-Strings in ein RootObject innerhalb des Backends ist im Aktivitätsdiagramm (siehe Abb. 31) separat dargestellt. Sollte bei der Umwandlung ein Fehler auftreten, zeigt Abbildung 32 die Reaktion des Frontends.

[15] Bibliothek zum Generieren und Einlesen von XML-Strings

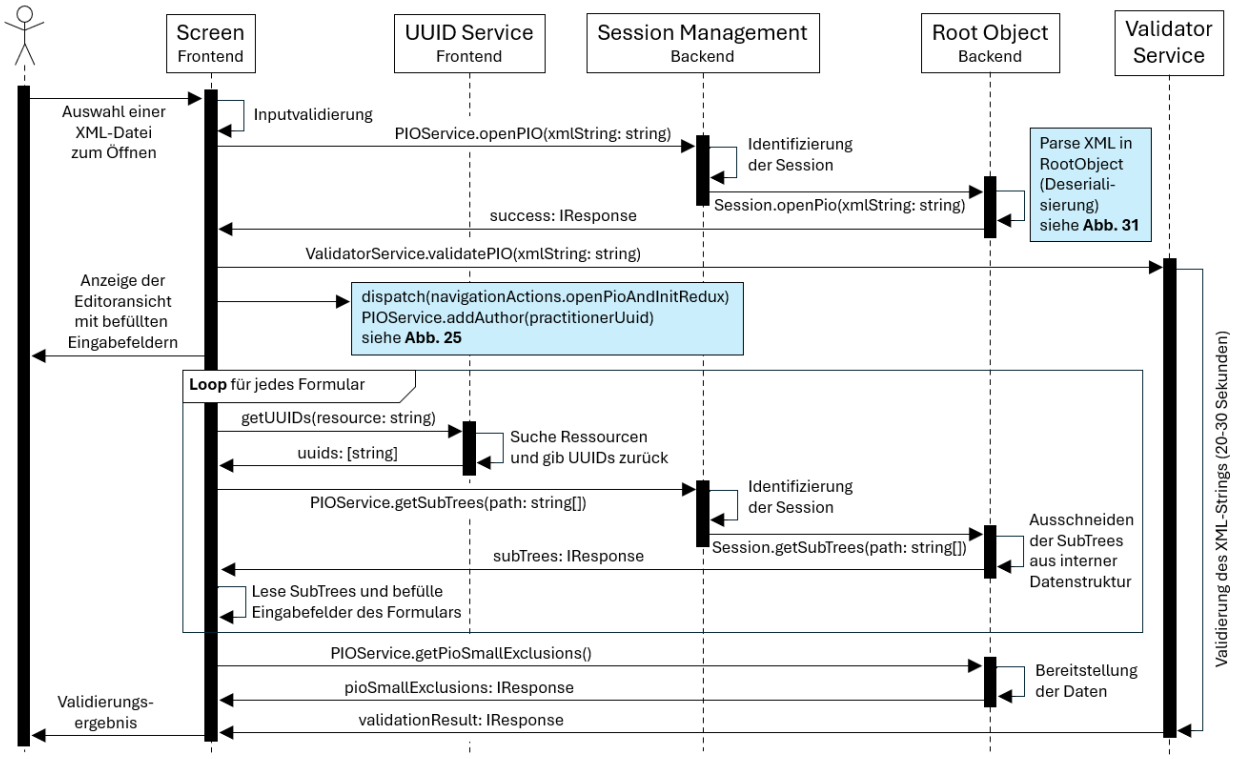

Abbildung 30: *Szenario „Import eines PIO-ULBs im XML-Format“ als UML-Sequenzdiagramm*

Wenn bei der Umwandlung im Backend kein Fehler auftritt, wird das Frontend über die erfolgreiche Interpretierung des XML-Strings informiert (siehe Abb. 30). Daraufhin wird der XML-String an den Validator Service gesendet, dessen Antwort erst nach 20 bis 30 Sekunden erwartet wird. Außerdem werden diverse Operationen zur Initialisierung des Redux Stores und des UUID Services ausgeführt sowie der angemeldete Nutzer als Autor hinterlegt. Dieser Prozess ist identisch mit dem Prozess aus Abbildung 25 und wird deshalb in Abbildung 30 nicht mehr dargestellt. Anschließend öffnet sich die Editoransicht.

Sobald die Editoransicht für den Nutzer sichtbar ist und somit die Eingabefelder im Browserfenster gerendert wurden, initialisieren sich alle Formulare selbständig. Wie die Initialisierung funktioniert, soll am Beispiel des Allergieformulars erklärt werden. Zuerst frägt das Formular beim UUID Service alle UUIDs ab, die mit Allergie-Ressourcen in Verbindung stehen. Existieren beispielsweise drei Allergie-Ressourcen im importierten PIO-ULB, werden drei Pfade nach der in Kapitel 6.3 erklärten Struktur generiert und an das Backend gesendet. Das Backend extrahiert die Daten, welche an den jeweiligen Pfaden abgelegt sind, und schickt sie als SubTrees an das Frontend zurück. Die drei Allergie-SubTrees werden im Formular gespeichert und ausgelesen. Die Eingabefelder in der Editoransicht werden anschließend mit den ausgelesenen Daten befüllt, sodass letztendlich das Allergieformular ein Akkordeon Menü mit drei Allergieeinträgen anzeigt.

Zum Schluss werden alle Daten vom Backend abgefragt, die nicht Teil des *PIO-Smalls* sind und daher keine eigenen Eingabefelder in der Editoransicht besitzen. Diese Daten werden unstrukturiert in einem dritten Tab (siehe Abb. 8) angezeigt.

Analog zum Szenario „Öffnen eines leeren PIO-ULBs“ (siehe Abb. 25) wird eine Flag *„imported“* im Redux Store gesetzt. Somit wird gespeichert, dass es sich um ein importiertes

PIO-ULB handelt, und der Validierungsbutton angezeigt werden soll. Das Speichern der Flag *„imported“* ist in Abbildung 30 nicht dargestellt.

Der bisher beschriebene Prozess besitzt eine Laufzeit von ungefähr zwei Sekunden. Nach diesen zwei Sekunden kann der Nutzer alle Funktionen der Editoransicht uneingeschränkt nutzen, abgesehen von der Validierungsfunktion. Das Validierungsergebnis kann erst nach ungefähr 20 bis 30 Sekunden eingesehen werden.

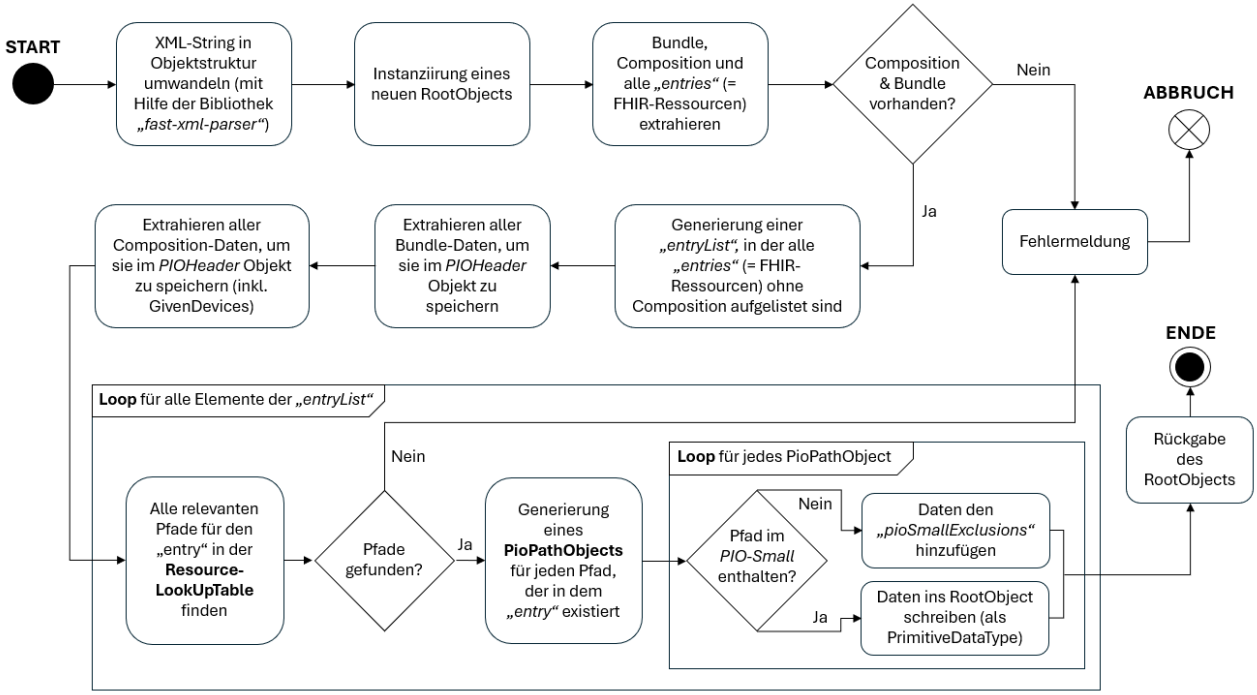

Abbildung 31: *Importprozess eines XML-Strings im Backend als UML-Aktivitätsdiagramm (Deserialisierungslogik)*

Abbildung 31 beschreibt detailliert, wie ein XML-String in ein RootObject konvertiert wird (**Deserialisierungslogik**). Hierbei werden nur Prozesse im Backend betrachtet. Wenn der Nutzer ein PIO-ULB importiert, wird der ausgelesene XML-String an das Backend geschickt, welches die statische Methode *„readXML()“* des RootObjects aufruft. Mit dem Aufruf der Methode startet der Prozess, der in Abbildung 31 dargestellt ist.

Im ersten Prozessschritt wird der XML-String mit Hilfe der Bibliothek *„fast-xml-parser“* in eine Objektstruktur umgewandelt. Eine objektbasierte Struktur erlaubt einen einfacheren Datenzugriff als textbasierte Quellen. Anschließend wird ein RootObject instanziiert, welches später mit den Daten aus dem PIO-ULB befüllt wird.

Im nächsten Schritt wird die Objektstruktur, welche den PIO-ULB abbildet, in relevante Einzelteile zerlegt. Es werden das *Bundle*, die *Composition* und alle weiteren *Entries* extrahiert und separat für die Weiterverarbeitung gespeichert. Das *Bundle* beinhaltet alle FHIR-Ressourcen als sogenannte ***Entries*** und bildet in seiner Gesamtheit ein FHIR-Dokument ab. Die *Composition* ist als Inhaltsverzeichnis des FHIR-Dokuments zu verstehen und ist auch als *Entry* im *Bundle* enthalten. Falls das *Bundle* oder die *Composition* nicht existieren, wird der Prozess mit einer Fehlermeldung abgebrochen.

Wenn das *Bundle* und die *Composition* vorhanden sind, wird der nächste Prozessschritt ausgeführt. Es wird eine *„entryList“* generiert, die alle *Entries* (ohne *Composition*) in einem Array sammelt. In die Liste werden also alle *Entries* aufgenommen, die relevante Patientendaten enthalten und somit Eingabefelder im Editor besitzen. Anschließend werden alle Informationen aus dem *Bundle* und der *Composition* ausgelesen und in dem *PIOHeader* Objekt, welches vom RootObject instanziiert wird (siehe Abb. 18), gespeichert. Diese Informationen werden im Kontext dieser Dokumentation als Header-Daten bezeichnet, da es sich um IDs, Zeitstempel, Titel des FHIR-Dokuments usw. handelt.

Eine Besonderheit besteht darin, dass die *Composition* auflistet, welche Geräte bzw. Hilfsmittel dem Patienten bei der Überleitung mitgegeben wurden. Da die FHIR-Ressourcen, die ein Gerät oder ein Hilfsmittel abbilden, keine Möglichkeit bieten, das Gerät bzw. das Hilfsmittel als „mitgegeben“ zu markieren, werden alle mitgegebenen Geräte und Hilfsmittel in der *Composition* mittels ihrer UUID referenziert. Auch diese Information wird extrahiert und im *PIOHeader* Objekt abgespeichert.

Der nächste Prozessschritt wird durch eine Iterationsschleife beschrieben, welche für jedes Element in der *„entryList“* ausgeführt wird. Ein Schleifendurchlauf beginnt mit dem Heraussuchen aller Pfade aus der *ResourceLookUpTable* (siehe Tab. 2), welche für den aktuellen *Entry* relevant sind. Wenn keine passenden Pfade gefunden werden können, kommt es zum Prozessabbruch.

Anschließend wird der gesamte *Entry* durchlaufen und für jeden Pfad, an dem Daten gespeichert sind, ein **PioPathObject** generiert. Ein *PioPathObject* enthält den Pfad sowie weitere Informationen über die dort gespeicherten Daten. Zum Beispiel wird hinterlegt, um welchen Datentyp es sich handelt und ob der Pfad zum *PIO-Small* gehört oder nicht. Da im XML-String alle Informationen über verwendete Datentypen verloren gegangen sind, müssen diese Informationen aus der *ResourceLookUpTable* entnommen werden.

Im Anschluss wird durch alle generierten *PioPathObjects* iteriert. Entweder werden die Daten als *PrimitiveDataType* (siehe Kapitel 9.1) in das RootObject geschrieben oder bei den *„pioSmallExclusions“* hinzugefügt, falls der Pfad nicht Teil des *PIO-Smalls* ist.

Sobald die gesamte *„entryList“* durchlaufen wurde, gibt die statische *„readXML()“* Methode das RootObject zurück, welches alle Daten aus dem XML-String beinhaltet.

Sollte im Backend ein Fehler auftreten und der Prozess abgebrochen werden, zeigt Abbildung 32 das dynamische Verhalten des Frontends:

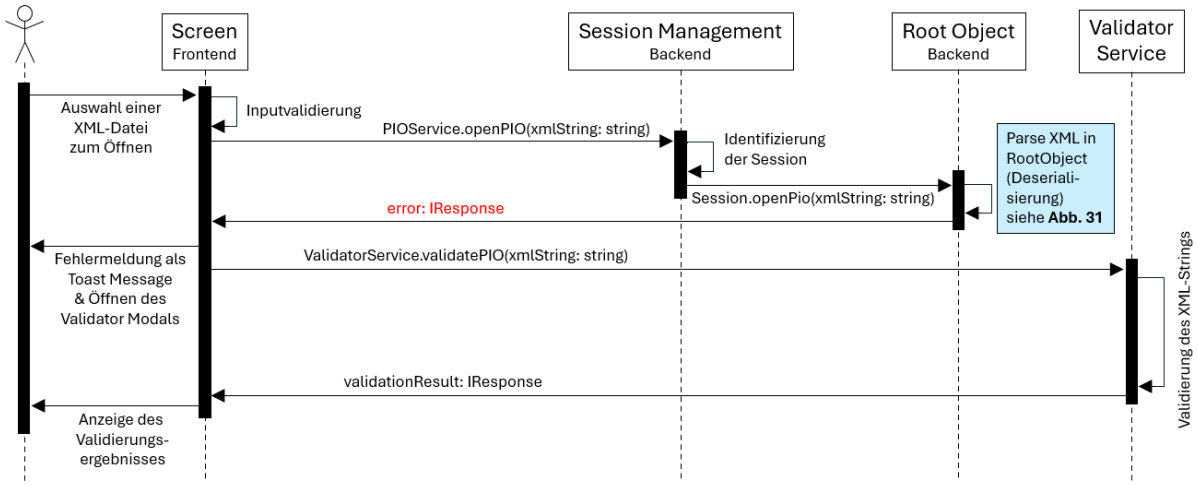

Abbildung 32: *Szenario „fehlerhafter Import einer XML-Datei“ als UML-Sequenzdiagramm*

8. Verteilungssicht

In diesem Kapitel wird die technische Infrastruktur des PIO-ULB Editors betrachtet. Dabei wird ausschließlich auf die *„localVersion“* (siehe Tab. 4) eingegangen, welche für die lokale Installation im Netzwerk einer Pflegeeinrichtung oder eines Krankenhauses vorgesehen ist. Abbildung 33 zeigt alle Komponenten des PIO-ULB Editors als UML-Verteilungsdiagramm:

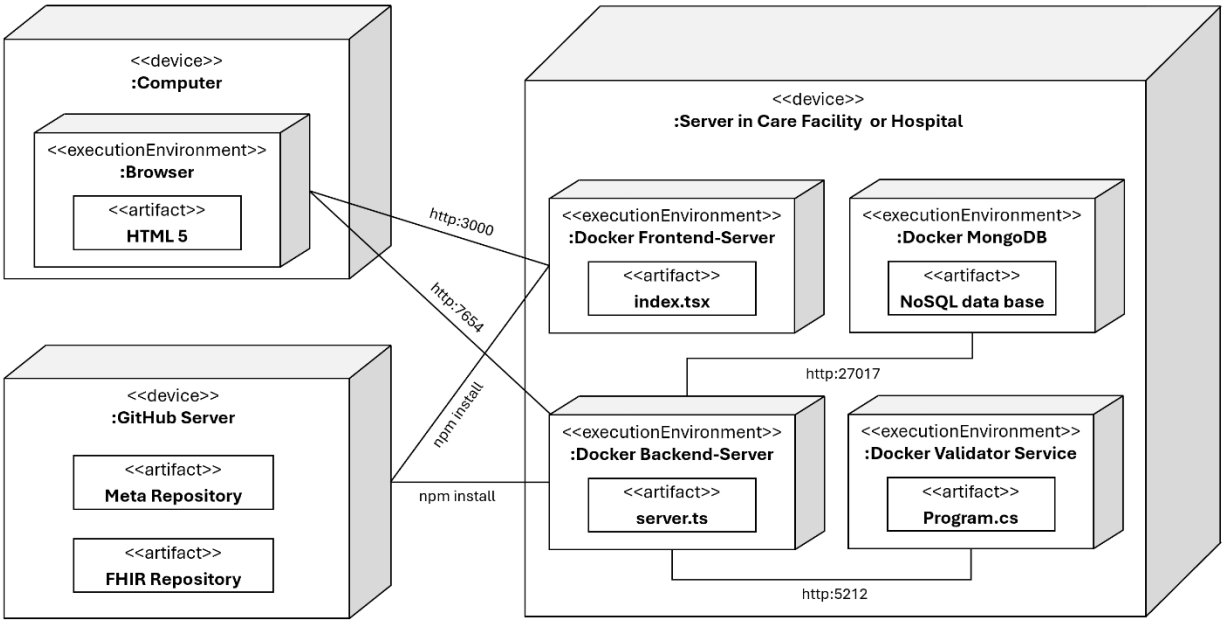

Abbildung 33: *UML-Verteilungsdiagramm des PIO-ULB Editors im lokalen Netzwerk einer Einrichtung*

Alle benötigten Komponenten werden als Docker Container bereitgestellt und müssen im lokalen Netzwerk der Einrichtung laufen, um den PIO-ULB Editor für das Pflegepersonal nutzbar zu machen. Das Pflegepersonal kann über einen beliebigen Computer im Netzwerk der Einrichtung den PIO-ULB Editor mit Hilfe des Browsers aufrufen. Das Meta- und FHIR-Repository werden sowohl im Frontend als auch im Backend benötigt und als NPM-Package von einer GitHub-Registry bereitgestellt. Dort ist auch der Quellcode des gesamten PIO-ULB Editors hinterlegt.

Das gesamte User Interface (React Applikation) wird vom Docker Container *Frontend-Server* über den Port 3000 als HTML Element zur Verfügung gestellt, welches im Browser dargestellt werden kann. Alle Anfragen an das Backend werden über den Port 7654 als HTTP-Request gesendet. Die Datenbank für das Adressbuch und der Validator Service laufen in einem eigenen Docker Container, welche direkt vom Backend angesprochen werden.

Auf GitHub wurden insgesamt sechs Repositories veröffentlicht [16]. Fünf Repositories beinhalten den Quellcode der fünf Subsysteme, die in Kapitel 6 bereits erklärt wurden (Frontend, Backend, Validator Service, Meta- und FHIR-Repository). Das Meta- und FHIR-Repository werden zusätzlich als NPM-Package bereitgestellt. Die Repositories, welche

[16] GitHub, *Technische Hochschule Augsburg – Institut für agile Softwareentwicklung*, URL: https://github.com/THAias [zuletzt aufgerufen: 10.03.2025].

den Quellcode für das Frontends, das Backend und den Validator Service enthalten, verfügen jeweils über eigene öffentliche Packages. Diese Packages stellen die entsprechenden Docker-Images zur Verfügung, wobei im Frontend und im Backend zwischen *„localVersion“* und *„webVersion“* unterschieden wird.

Um den PIO-ULB Editor schnell und einfach auf einem Computer zu installieren, existiert das sechste Repository *„PIO_Editor“*. Dieses Repository enthält eine *„docker-compose.yml“* Datei. Bei einer korrekten Ausführung dieser Datei werden alle nötigen Docker-Images automatisch heruntergeladen und in einem Docker-Netzwerk gestartet, sodass der PIO-ULB Editor ohne weiteren Konfigurationsaufwand über den Browser aufgerufen werden kann. Es wird standardmäßig die *„localVersion“* installiert. Die Installationsanleitung ist in der *README-Datei* des Repositories zu finden und wird in Folgendem kurz dargestellt:

Kurzanleitung zur Installation des PIO-ULB Editors (für Windows):

1. *„PIO_Editor“* Repository von GitHub downloaden [17]

2. Docker Engine auf dem System installieren und starten

3. Die Datei *„docker-compose.yml“* aus dem *„PIO_Editor“* Repository mit Hilfe des Terminalbefehls `docker compose up -d` ausführen

4. Alle nötigen Docker Container werden automatisch heruntergeladen und gestartet (Internetverbindung benötigt)

5. Der PIO-ULB Editor (*„localVersion“*) ist nun unter der URL `http://localhost` im Browser erreichbar

Während der Installation werden bei Schritt 4 insgesamt fünf Docker Container heruntergeladen und gestartet. Vier dieser Docker Container sind in Abbildung 33 zu sehen (Frontend, Backend, Datenbank, Validator Service). Das Docker-Image für das Frontend, das Backend und den Validator Service wird von GitHub gezogen. Das Image der MongoDB Datenbank wird von Docker-Hub heruntergeladen. Der fünfte Docker Container ist ein **NGINX-Server**. Dieser Server fungiert als Reverse Proxy für alle Backend Anfragen und ist in Abbildung 33 nicht dargestellt. Der NGINX-Server übernimmt in der Softwarearchitektur keine außergewöhnlichen Aufgaben, sondern soll lediglich die Performance und die Sicherheit des PIO-ULB Editors steigern.

Um den PIO-ULB Editor in der *„webVersion“* zu starten, müssen zwei Zeilen der *„docker-compose.yml“* Datei angepasst werden. In Zeile 4 und 19 muss der Image-Tag *„:localVersion“* in *„:webVersion“* umbenannt werden.

[17] GitHub, *PIO_Editor*, URL: https://github.com/THAias/PIO_Editor [zuletzt aufgerufen: 25.03.2025].

9. Querschnittliche Konzepte

Dieses Kapitel befasst sich mit übergreifenden, generellen Konzepten und Lösungsansätzen, die in vielen Teilen der Softwarearchitektur einheitlich benutzt werden. Dabei wird auf folgende Konzepte eingegangen:

- PrimitiveDataType Klassen
- SubTree Klasse
- Struktur des Redux Stores
- Formulare zur Dateneingabe im Frontend

9.1 PrimitiveDataType Klassen

Die 12 PrimitiveDataType Klassen werden vom Meta-Repository bereitgestellt und finden sowohl im Frontend als auch im Backend Anwendung. Sie bilden Datentypen ab, die im FHIR Standard vorgesehen sind (z.B. String, DateTime, Code, Boolean, Integer, usw.). Im PIO-ULB Editor werden Patientendaten stehts als Instanz einer PrimitiveDataType Klasse gespeichert. Das heißt, dass die hierarchische Objektstruktur im Backend, welche ein PIO-ULB repräsentiert, stehts einen PrimitiveDataType auf der niedrigsten Hierarchiestufe hält (siehe Abb. 19). Folgende PrimitiveDataType Klassen wurden im PIO-ULB Editor implementiert und zu einem Typen zusammengefasst:

```
type PrimitiveDataTypes =

   StringPIO | CodePIO | UriPIO | UuidPIO | BinaryPIO |
   DateTimePIO | DatePIO | BooleanPIO | IntegerPIO |
   DecimalPIO | UnsignedIntegerPIO | MarkdownPIO
```

FHIR stellt zwar mehr primitive Datentypen zur Verfügung, es werden jedoch nur diese 12 Datentypen für den Überleitungsbogen benötigt. Alle PrimitiveDataTypes erben von einer abstrakten Mutterklasse. Die Vererbung ist beispielhaft für zwei Klassen (CodePIO und DatePIO) in Abbildung 34 dargestellt.

Jeder PrimitiveDataType hält ein Attribut *„_value“*. Dieses Attribut speichert die Patientendaten. Natürlich stellen die Klassen ebenso *get()* und *set()* Methoden bereit, mit denen auf die Patientendaten lesend und schreibend zugegriffen werden kann.

Außerdem sind zwei weitere Methoden vorhanden, die für die Serialisierungs- bzw. Deserialisierungslogik benötigt werden. Beim Exportieren eines PIO-ULBs muss das Root-Object im Backend in einen XML-String konvertiert werden. Deshalb ermöglicht die Methode *toString()* die Umwandlung eines PrimitiveDataTypes in einen String. Beim Import eines PIO-ULBs ist die umgekehrte Konvertierung nötig. Hier müssen die Patientendaten, die als String in der XML-Datei vorliegen, in den richtigen PrimitiveDataType umgewandelt werden. Hierfür steht die statische Methode *parseFromString()* zur Verfügung.

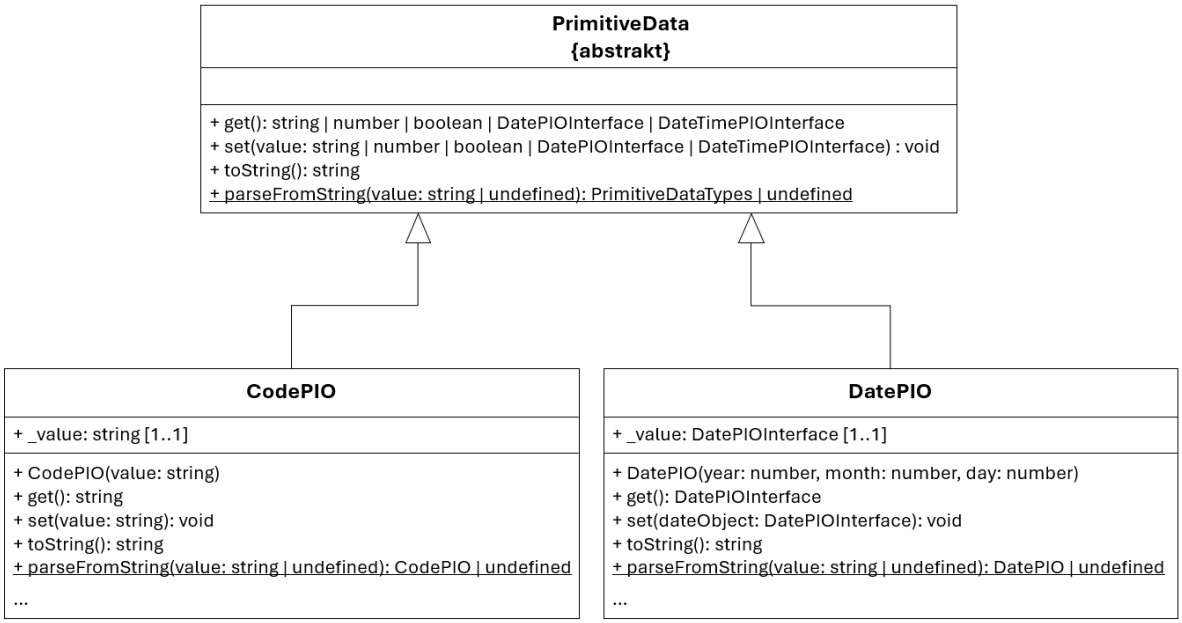

Abbildung 34: *UML-Klassendiagramm für zwei beispielhafte PrimitiveDataTypes (CodePIO und DatePIO)*

Weiterhin werden die PrimitiveDataTypes auch innerhalb der SubTree Klasse verwendet, um Patientendaten abzubilden. Die SubTree Klasse wird im nächsten Kapitel behandelt.

9.2 SubTree Klasse

Die SubTree Klasse wird vom Meta-Repository bereitgestellt und dient der Kommunikation zwischen Frontend und Backend. Werden Patientendaten zum Speichern an das Backend gesendet oder Patientendaten vom Backend abgefragt, geschieht dies immer als SubTree. Ein SubTree ist ein rekursiv aufgebautes Objekt, welches sich selbst mehrfach instanziieren kann (siehe Abb. 35, *Attribut: children*). Durch diese Verschachtelung entsteht eine hierarchische Baumstruktur, wobei an jedem Knotenpunkt ein PrimitiveDataType gespeichert werden kann (siehe Abb. 35, *Attribut: data*). Dadurch kann die Struktur eines PIO-ULBs als SubTree abgebildet werden.

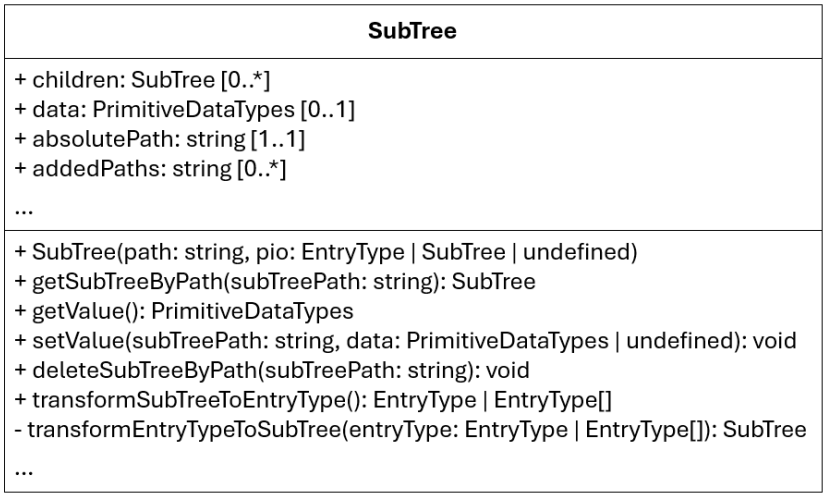

Abbildung 35: *UML-Klassendiagramm der SubTree Klasse*

Ein SubTree soll niemals die gesamte Struktur eines PIO-ULBs abbilden, sondern immer nur einen kleinen Teilausschnitt. Ein SubTree ermöglicht also das isolierte Abrufen und Manipulieren eines kleinen Teilausschnitts des PIO-ULBs (siehe Abb. 35, *Methoden: getValue & setValue*). So kann z.B. das *PatientCommunication* Formular im Frontend isoliert die gesprochenen Sprachen des Patienten abfragen, ohne dabei auf die gesamte FHIR Patienten-Ressource zuzugreifen. Weiterhin kann das Formular den SubTree isoliert manipulieren und gesprochene Sprachen hinzufügen oder entfernen. Die Manipulation des SubTrees hat keine direkten Auswirkungen auf die Daten im Backend. Erst wenn das Formular den SubTree zum Speichern an das Backend schickt, wird der Teilausschnitt, den der SubTree abbildet, durch den SubTree ersetzt.

Außerdem hat die SubTree Klasse zwei weitere Attribute (siehe Abb. 35). Das Attribut *„addedPaths“* speichert alle Manipulationen, die durch die Methode setValue() am SubTree vorgenommen wurden. Wenn der SubTree an das Backend geschickt wird, überprüft der *PathValidator* (siehe Abb. 17), ob die manipulierten Pfade konform zum PIO-ULB Standard sind. Das Attribut *„absolutePath“* muss in jedem SubTree angegeben sein, weil es genau beschreibt, an welcher Stelle der SubTree ausgeschnitten wurde. Ein SubTree, welcher die gesprochenen Sprachen des Patienten abbildet, könnte folgenden *„absolutePath“* besitzen:

```
dd023723-445f-4440-a474-e111016e3d81.KBV_PR_MIO_ULB_Patient.communication
```

Des Weiteren ist der Konvertierungsaufwand hervorzuheben, welcher durch die Verwendung von SubTrees entsteht. Weil die Daten im Backend als *„EntryType“* (siehe Abb. 19) vorliegen und ein SubTree eine andere Datenstruktur aufweist, muss bei jeder Erzeugung und bei jedem Abspeichern eines SubTrees eine Konvertierung stattfinden. Bei der Erzeugung eines SubTrees ruft der Konstruktor der Klasse die private *„transformEntryTypeToSubTree“* Methode auf, während bei einer Abspeicherung die *„transformSubTreeToEntryType“* Methode Anwendung findet.

9.3 Struktur des Redux Stores

Der Redux Store wird als globaler Zustandsspeicher im Frontend verwendet. Hier werden Navigationsinformationen der Applikation, Nutzerdaten, FHIR-Daten und die aktuelle Validierungs-ID gespeichert. In diesem Kapitel wird nur auf die Struktur des Redux Stores eingegangen. Die zugehörigen *Reducer* und *Actions* werden nicht erklärt, sind aber im Quellcode gut kommentiert. Abbildung 36 zeigt die Struktur des Redux Stores in der *„localVersion“* (siehe Tab. 4) mit beispielhaftem Inhalt:

```
authState:
   userData: { firstName: „Max“, lastName: „Mustermann“ }
   isLoggedIn: true
   autoLogin: true
   disclaimerAccepted: true
navigationState:
   screen: „2“
   openPio: true
   openPioType: „imported“
   collapsedMenu: false
   exportPio: undefined
patientState:
   patientBirthdate: „1949-08-17T10:00:00.000Z“
   patientName: { prefix: „Dr.“, familyName: „Müller“, givenName: „Peter“, … }
   patientGender: „Männlich“
patientExtensionState:
   0: { url: „https:// … Religion“, value: „katholisch“, dataType: „String“ }
   1: { url: „https:// … Interpreter_Required“, value: „true“, dataType: „Boolean“ }
   2: { url: „https:// … Notes_For_Communication“, value: „Spricht nur Türkisch“, dataType: „String“ }
contactPersonState:
   0: { id: …, role: …, gender: …, name: …, address: …, telecom: … }
   1: { id: …, role: …, gender: …, name: …, address: …, telecom: … }
practitionerState:
   0: { id: …, additionalInfo: …, organization: …, qualification: …, role: …, speciality: …, … }
   1: { id: …, additionalInfo: …, organization: …, qualification: …, role: …, speciality: …, … }
   2: { id: …, additionalInfo: …, organization: …, qualification: …, role: …, speciality: …, … }
organizationState:
   0: { id: …, name: …, type: …, identifier: …, address: …, telecom: … }
   1: { id: …, name: …, type: …, identifier: …, address: …, telecom: … }
   2: { id: …, name: …, type: …, identifier: …, address: …, telecom: … }
   3: { id: …, name: …, type: …, identifier: …, address: …, telecom: … }
mockedDataBase: []
validationId: „a067095e-bfec-407a-abfd-7b989abd9226“
```

Abbildung 36: *Struktur des Redux Stores in der „localVersion“ mit beispielhaftem Inhalt*

Der Zustandsspeicher *„authState“* beinhaltet Nutzer- und Authentifizierungsdaten. Neben dem Vor- und Nachnamen des Nutzers werden hier auch gewisse Status-Flags gespeichert. Der Auto-Login ist nur in der *„localVersion“* aktiv und bewirkt einen automatischen Login der zuletzt verwendeten Session. Weiterhin wird gespeichert, ob der Disclaimer vom Nutzer bereits akzeptiert wurde. Sobald die Flag *„disclaimerAccepted“* auf *false* gesetzt wird, öffnet sich der Disclaimer als Popup Fenster und muss akzeptiert werden, bevor man den PIO-ULB Editor uneingeschränkt nutzen kann.

Der Speicher *„naviagtionState“* merkt sich Navigationsinformationen. Je nachdem welche Zahl unter *„screen“* hinterlegt ist, wird im PIO-ULB Editor ein anderer Bildschirm angezeigt. Im Beispiel ist die Zahl zwei hinterlegt (siehe Abb. 36), was bedeutet, dass die Editoransicht zu sehen ist. Die Flag *„openPio“* speichert, ob aktuell ein PIO-ULB zur Bearbeitung geöffnet ist (*true*) oder nicht (*false*). Wenn ein PIO-ULB geöffnet ist, klassifiziert die Flag *„openPioType“*, ob es sich um einen importierten PIO-ULB handelt (*„imported“*) oder um einen neu geöffneten PIO-ULB (*„new“*). Diese Unterscheidung ist wichtig, da nur bei einem importierten PIO-ULB der Validierungs-Button im Menü der Tab-Bar erscheinen soll. Weiterhin speichert der Zustand *„collapsedMenu“*, ob das Seitenmenü in der Editoransicht eingeklappt ist (*true*) oder nicht (*false*), und *„exportPio“* enthält im Normalfall *undefined*. Während eines laufenden Exportprozess gibt der State *„exportPio“* an, ob eine XML-Datei (*„xml“*) oder eine PDF-Datei (*„pdf“*) exportiert wird. Sobald der State einen gültigen String-Wert enthält (also *„xml“* oder *„pdf“*) werden in der Editoransicht alle Akkordeonmenüs aufgeklappt, damit eine vollständige Validierung aller Eingabefelder vorgenommen werden kann.

Der *„patientState“* speichert die wichtigsten Patientenstammdaten (Name, Geburtsdatum und Geschlecht), damit in der Editoransicht die Überschrift mit diesen Daten befüllt werden kann (siehe Abb. 21, Ebene 3, *HeaderPatientName.tsx*).

Die nachfolgenden Zustände speichern hauptsächlich FHIR-Daten. So beinhaltet *„patientExtensionState“* alle Extensions der FHIR-Patient Ressource. Dies ist nötig, weil die Religionszugehörigkeit in einem anderen Formular abgefragt wird, wie die sprachbezogenen Informationen. Da der SubTree aus dem einen Formular, die Extensions aus dem anderen Formular überschreiben würde, greifen beide Formulare auf die Extensions im globalen Zustandsspeicher zu, um Dateninkonsistenzen zu vermeiden.

Die nächsten drei Zustandsspeicher *„contactPersonState“*, *„practitionerState“* und *„organizationState“* beinhalten Daten kompletter FHIR-Ressourcen. Hier werden alle Kontaktpersonen, alle behandelnden Personen und alle Organisationen aus dem Adressbuch gespeichert. Das ist nötig, weil diese Ressourcen innerhalb des PIO-ULBs referenziert werden können. So kann der Nutzer zum Beispiel beim Anlegen einer neuen Diagnose einen Diagnosesteller angeben. Als Diagnosesteller wird eine Referenz auf eine behandelnde Person (= Practitioner) hinterlegt. Deshalb wird ein Drop-Down-Menü mit allen behandelnden Personen aus dem Redux Store vorinitialisiert, sodass der Nutzer den Diagnosesteller nur noch aus dem Drop-Down-Menü auswählen muss.

Der Zustandsspeicher *„mockedDataBase“* wird nur für die *„webVersion“* verwendet und speichert alle Adressbucheinträge im Frontend. In der *„localVersion“* bleibt dieser Zustandsspeicher leer (siehe Tab. 4). Zu guter Letzt wird die ID des aktuellsten Validierungsprozesses gespeichert, damit die einzelnen Validierungsprozesse im Frontend richtig zugeordnet werden können. Dies ist nötig, da ein Validierungsprozess bis zu 30 Sekunden dauern kann und der Nutzer innerhalb der 30 Sekunden einen zweiten Validierungsprozess anstoßen könnte.

9.4 Formulare zur Dateneingabe im Frontend

In der Editoransicht des PIO-ULB Editors sind thematisch zusammengehörende Eingabefelder als Formular zusammengefasst, wobei sich deren Struktur fast ausschließlich von den verwendeten FHIR-Ressourcen ableitet. Jedes Formular speichert die Nutzereingaben in Form eines SubTrees und sendest diese selbständig an das Backend (siehe Abb. 26). Im PIO-ULB Editor gibt es zwei unterschiedliche Formulararten:

- Formulare, die eine FHIR-Ressource abbilden, welche laut Spezifikation maximal einmal im PIO-ULB existieren dürfen
- Formulare, die eine FHIR-Ressource abbilden, welche laut Spezifikation mehrfach im PIO-ULB existieren kann und deshalb als Akkordeon Menü dargestellt wird

Zum Beispiel darf die Ressource zur Abbildung des Barthelindexes im Tab „Pflegerische Infobausteine“ maximal einmal im gesamten PIO-ULB vorkommen. Deshalb enthält das Formular unmittelbar Eingabefelder ohne Akkordeonmenü. Weiterhin erlaubt die PIO-ULB Spezifikation die mehrfache Verwendung von Risiko-Ressourcen. Aus diesem Grund

zeigt das entsprechende Formular im Tab „Pflegerische Infobausteine“ ein Akkordeonmenü. Jeder Eintrag im Akkordeon Menü repräsentiert also eine Risiko-Ressource. Im Folgenden sollen beide Formulare genauer betrachtet werden:

Abbildung 37: *Das Formular „Barthelgesamtindex“ im Tab „Pflegerische Infobausteine“ des PIO-ULB Editors*

Das **Formular „Barthelgesamtindex“** ist eine React-Komponente, welche zwei Eingabefelder und eine Infobox enthält. Im Quellcode wird das Formular durch die Datei *„Barthel-Form.tsx“* beschrieben. Abbildung 37 zeigt das Formular aus Nutzerperspektive.

Neben den graphischen UI-Elementen umfasst das Formular einen lokalen State, welcher den benötigten SubTree hält, sowie zusätzliche Logik zum Auslesen und Manipulieren des SubTrees. Wird ein PIO-ULB importiert, initialisiert sich das Formular selbständig über einen *useEffect-Hook* (siehe Kapitel 7.7). Dabei wird der Teilausschnitt, welcher den Barthelgesamtindex repräsentiert, als SubTree aus dem Backend geladen und im lokalen State gespeichert. Sobald der SubTree im Formular vorliegt, wird der Index und das Feststellungsdatum ausgelesen und in die Eingabefelder geschrieben. Wenn der Nutzer Daten editiert und anschließend aus dem Eingabefeld herausklickt, wird das *onBlur-Event* ausgelöst, welches wiederum die *onFinish()* Methode aufruft. Diese Methode ist in jedem Formular ausnahmslos vorhanden. Sie beschreibt den hinterlegten SubTree mit den neuen Daten und sendet den SubTree direkt an das Backend (siehe Kapitel 7.4). Jedes andere Formular weist eine analoge Struktur auf. Der *useEffect-Hook* ist stets für das Auslesen des SubTrees und die Initialisierung der Eingabefelder zuständig, während die *onFinisch()* Methode neue Daten in den SubTree schreibt und an das Backend schickt.

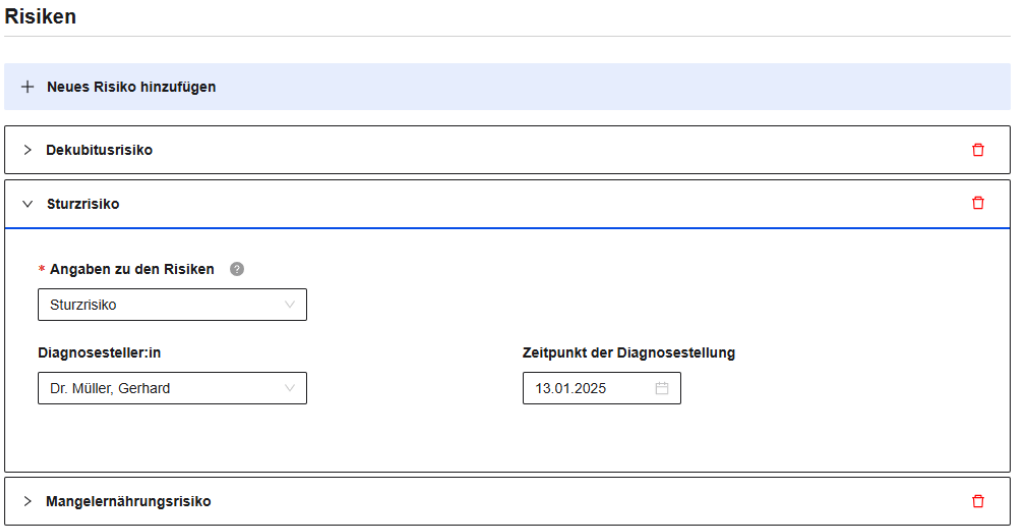

Abbildung 38: *Das Formular „Risiken“ im Tab „Pflegerische Infobausteine“ des PIO-ULB Editors*

Das **Formular „Risiken“** ist eine React-Komponente, welche ein Akkordeon Menü beinhaltet, weil ein Patient mehreren Risiken ausgesetzt sein kann. Im Quellcode wird das Formular durch die Datei *„RiskForm.tsx“* beschrieben. Abbildung 38 zeigt das Formular aus Nutzerperspektive.

Formulare, die ein Akkordeon Menü darstellen, verfügen ebenfalls über einen lokalen State, in dem ein Array aus SubTrees gehalten wird, wobei jeder SubTree einem Risiko entspricht. Der *useEffect-Hook* und die *onFinisch()* Methode müssen demnach über das Array iterieren und mehrere SubTrees auslesen oder manipulieren.

Außerdem kann jeder Eintrag im Akkordeon Menü ausgeklappt werden, wobei alle Einträge die gleichen Eingabefelder enthalten. Über den Button *„Neues Risiko hinzufügen“* kann dem Akkordeon Menü ein neuer Eintrag hinzugefügt werden. Die Daten für den neuen Eintrag werden in einem Modal abgefragt, bevor das neue Risiko im Akkordeon Menü angezeigt wird (siehe Abb. 39). Das Modal enthält ebenfalls die gleichen Eingabefelder wie alle anderen Einträge im Akkordeon Menü.

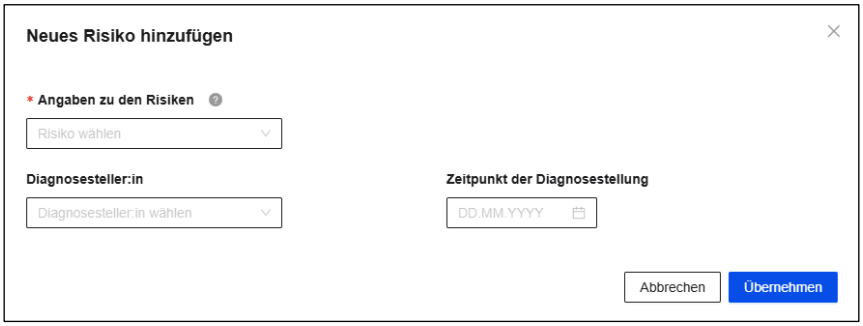

Abbildung 39: *Modal zum Hinzufügen eines neuen Risikos*

Die Häufigkeit, mit der identische Eingabemasken im Rahmen von Akkordeon Menüs auftreten, legt nahe, diese Eingabemasken als wiederverwendbare Komponente zu implementieren. Deshalb gibt es für jedes Akkordeon Menü eine Wrapper Komponente, welche alle UI-Elemente der wiederkehrenden Eingabemaske ohne jegliche Logik bündelt. Im Fall des Risiko Formulars stellt die Datei *„RiskWrapper.tsx“* die wiederkehrende Eingabemaske bereit. Dieser Wrapper kann sowohl dem Akkordeon Menü als auch dem Modal übergeben werden. Das Akkordeon Menü selbst ist in der Komponente *„MultiWrapper.tsx“* enthalten, sodass das Risiko Formular lediglich den Multi-Wrapper instanziieren und den Risk-Wrapper übergeben muss. Abbildung 21 verdeutlicht den Aufbau der Formulare visuell.

10. Architekturentscheidungen

In diesem Kapitel sollen wichtige Architekturentscheidungen, die zentrale Bedeutung für das Gesamtsystem haben, erläutert und begründet werden. Einige dieser Architekturentscheidungen können im Nachhinein als unpraktikabel eingestuft werden und sollten in einer Folgeimplementierung des PIO-ULB Editors optimiert werden. Alle unpraktikablen Entscheidungen werden in Kapitel 11 erneut aufgegriffen und als technische Schulden kritisch diskutiert. Tabelle 9 listet alle Architekturentscheidungen in diesem Kapitel auf:

Tabelle 9: *Wichtige Architekturentscheidungen und deren Auswirkungen auf die Softwarearchitektur*

Benennung	Beschreibung	Praktikabilität	Resultierende tech. Schulden
Webanwendung	Bereitstellung des Editors als Webanwendung mittels Server	praktikabel	nein
Generisches Backend	Backend kennt und validiert die Struktur eines PIO-ULBs nicht	unpraktikabel [18]	nein
PrimitiveData-Types	Abbildung jeder Nutzereingabe als PrimitiveDataType Instanz	unpraktikabel	ja
SubTrees	Kommunikation zwischen Front- und Backend mittels Teilausschnitten	bedingt praktikabel	ja
onBlur-Event	Speicherung der Nutzereingaben durch „Herausklicken“	praktikabel	nein
Login-Verhalten	Anmeldung mittels Vor- und Nachname	praktikabel	ja
Adressbuch	Persistente Speicherung von FHIR-Organisationsressourcen	praktikabel	ja

Der PIO-ULB Editor wurde als **Webanwendung** entwickelt, bei der ein oder mehrere Frontend Clients auf einen zentralen Backend-Server zugreifen. Das Frontend ist als React Anwendung implementiert worden, welche unter einer lokalen URL zu erreichen ist und deshalb Servereigenschaften aufweist. Streng genommen handelt es sich jedoch nicht um einen Server, sondern um statisches HTML, CSS und JavaScript, das von einem Webserver ausgeliefert wird. Es können also beliebig viele Computer im lokalen Netzwerk auf dieselbe URL und somit auf das Frontend des PIO-ULB Editors zugreifen. Jeder Computer erhält jedoch eine eigene Instanz des statischen HTML-Codes, weshalb jeder Computer als isolierter Client betrachtet werden kann. Diese Clients greifen letztendlich auf eine zentrale Backend-Instanz zu, welche als Server betrachtet werden kann.

Diese Architekturentscheidung wurde getroffen, um eine möglichst einfache Bereitstellung des PIO-ULB Editors zu gewährleisten. Da die Client-Computer das Frontend über eine URL abrufen, ist es nicht nötig, auf jedem Computer Software zu installieren. Es genügt, einen Computer an das lokale Netzwerk anzuschließen und der PIO-ULB Editor kann ohne Installationsaufwand über einen beliebigen Browser benutzt werden. Außerdem ermöglichen Webanwendungen eine zentrale Verwaltung der Software. Zum Beispiel

[18] Im Fall des PIO-ULB Editors, war das generische Backend unpraktikabel. Würden aber verschiedene Applikationen, die unterschiedliche MIOs und PIOs erzeugen können, auf das selbe generische Backend zugreifen, könnte ein generisches Backend praktikabel sein.

genügt es, mögliche Updates auf dem Server durchzuführen und nicht auf jedem einzelnen Gerät.

Der Backend-Server wurde als **generisches Backend** konzipiert. Ein generisches Backend verarbeitet und speichert Daten, ohne dessen Struktur zu validieren. Im Fall des PIO-ULB Editors hat dies folgende Bedeutung: Daten, die der Nutzer in ein Formular einträgt, werden im Frontend mit Strukturinformationen (also mit einem XML-Pfad, siehe Kapitel 6.3) verknüpft und gebündelt an das Backend gesendet. Diese Bündelung der eingetragenen Daten und der Strukturinformationen findet in den SubTree Objekten statt. Das Backend speichert die Nutzereingabe am übermittelten XML-Pfad ab, ohne zu prüfen, ob der XML-Pfad spezifikationskonform ist. Deshalb verlagert sich die gesamte strukturbezogene Logik, welche den XML-Aufbau eines PIO-ULBs beschreibt, ins Frontend. Das Backend bleibt dadurch sehr schlank während die Komplexität des Frontends zunimmt.

Diese Architekturentscheidung wurde getroffen, weil ein generisches Backend wartungsarm ist. Sollte sich die PIO-ULB Spezifikation strukturell ändern, wären nur Anpassungen am Frontend nötig. Außerdem ermöglicht ein generisches Backend die Wiederverwendung für andere MIO- oder PIO-Typen. Durch die Verknüpfung mit einem anderen Frontend sowie geringfügigen Anpassungen im Backend kann ein Editor für ein anderes MIO oder ein anderes PIO entstehen. Die Robustheit gegenüber Spezifikationsänderungen und die Kompatibilität mit anderen MIOs/PIOs sind starke Argumente für ein generisches Backend, dennoch wird diese Architekturentscheidung im Kontext des PIO-ULB Editors als unpraktikabel eingestuft und in Kapitel 11 genauer betrachtet. Daraus resultieren jedoch keine technischen Schulden.

Des Weiteren wurden alle verwendeten FHIR Datentypen als **PrimitiveDataTypes** (siehe Kapitel 9.1) im PIO-ULB Editor umgesetzt. Jede Nutzereingabe wird als Instanz einer PrimitiveDataTypes Klasse im Backend gespeichert. Diese Architekturentscheidung wurde getroffen, um den Editor möglichst analog zur FHIR-Spezifikation und dessen vorgeschriebene Datentypen zu entwerfen. Die Datentypkonformität zur FHIR-Spezifikation klingt zuerst plausibel, hat sich im Laufe der Entwicklungsarbeit jedoch als unpraktikabel herausgestellt und zu deutlichem Mehraufwand geführt (siehe Kapitel 11).

Die **SubTrees** (siehe Kapitel 9.2) dienen der Kommunikation zwischen Front- und Backend. Sie bilden einen Teilausschnitt der XML-Datenstruktur ab und enthalten die Nutzereingaben aus dem Frontend. Grundsätzlich ist die Idee, nur mit relevanten Teilen der XML-Gesamtstruktur zu arbeiten, sehr effizient. Dadurch wird ermöglicht, das jedes Formular im Frontend seine eigenen Daten speichert, verwaltet und manipuliert. Der rekursive Aufbau der SubTree Klasse führt jedoch zu unnötig vielen Konvertierungsschritten. Deshalb wird die Implementierung der SubTrees als unpraktikabel eingestuft (siehe Kapitel 11), während die grundsätzliche Idee der Datenstückelung unbedingt beibehalten werden sollte.

Weiterhin verwendet jedes Formular im Frontend das **onBlur-Event**, um Nutzereingaben zu speichern. Jedes Mal, wenn der Nutzer aus einem Eingabefeld „herausklickt“, wird das Event getriggert und alle Daten des Formulars im Backend gespeichert. Diese Architekturentscheidung wurde getroffen, weil dadurch alle Nutzereingaben direkt nach der

Eingabe gespeichert werden, ohne dass der Nutzer eine aktive Handlung ausführen muss. Das Risiko eines Datenverlustes durch Absturz des Frontend-Clients oder eines automatischen Logouts aufgrund Inaktivität wird somit ausgeschlossen. Die Architekturentscheidung war demnach praktikabel und verursacht keine technischen Schulden.

Beim **Login** eines Nutzers wird lediglich der Vor- und Nachname abgefragt. Melden sich zwei Nutzer mit demselben Namen an, kommt es zu ungewollten Nebeneffekten, da beide Clients auf derselben Datenbasis arbeiten. Diese Besonderheit des Login-Verhaltens wird jedoch bewusst toleriert, da es sich beim PIO-ULB Editor um einen Prototyp handelt. Die Implementierung eines robusten Usermanagements macht erst in einer kommerziellen Entwicklung Sinn. Außerdem sollte die Funktionalität des PIO-ULB Editors ohnehin in die Dokumentationssoftware der pflegerischen Einrichtungen integriert werden, um die Anzahl an verwendeten Stand-Alone-Systemen zu reduzieren. In diesem Fall würde das bereits existierende Usermanagement der Dokumentationssoftware eine eigene Nutzerverwaltung des PIO-ULB Editors überflüssig machen. Das Login-Verhalten des PIO-ULB Editors ist also eine Architekturentscheidung, die fehleranfällig ist, im Rahmen der prototypischen Entwicklung jedoch praktikabel war. Die daraus resultierenden technischen Schulden werden in Kapitel 11 erläutert.

Das **Adressbuch** ermöglicht das persistente Speichern von FHIR-Organisationsressourcen. Die gespeicherten Organisationen können vom Nutzer in der Editoransicht des PIO-ULB Editors ausgewählt werden. Diese Architekturentscheidung wurde getroffen, um dem Nutzer unnötigen Arbeitsaufwand zu ersparen. Daten von bereits bekannten Pflegeheimen oder Krankenhäusern müssen nicht in jedem PIO-ULB erneut eingetragen, sondern können aus einem Drop-Down Menü ausgewählt werden. Aktuell speichert das Adressbuch lediglich Organisationsressourcen. Im Quellcode wurde jedoch berücksichtigt, dass in Zukunft auch andere Ressourcentypen im Adressbuch abgelegt werden können. Beispielsweise könnte es sinnvoll sein, Practitioner-Ressourcen (= behandelnde Person wie Arzt oder Pflegekraft) ebenfalls persistent zu speichern. Diese Architekturentscheidung ist für die Gebrauchstauglichkeit der Software ein klarer Vorteil, jedoch ergeben sich gewisse Probleme, wenn man FHIR-Ressourcen PIO-ULB übergreifend speichern will. Diese Probleme werden in Kapitel 11 als technische Schulden genauer betrachtet.

11. Technische Schulden

In diesem Kapitel werden alle Architekturentscheidungen, die in Kapitel 10 als unpraktikabel oder als Ursache für technische Schulden identifiziert wurden, genauer betrachtet. Einige technische Schulden werden bewusst toleriert, weil der PIO-ULB Editor nur als Prototyp konzipiert wurde.

Gewöhnlicherweise implementiert das Frontend lediglich die Benutzeroberfläche, während das Backend die Datenverarbeitung und Speicherung übernimmt. Diese übliche Aufgabenteilung kann im PIO-ULB Editor nicht eingehalten werden, da Logik zur Datenverarbeitung ins Frontend wandern muss, wenn das **Backend generisch** implementiert wird. Die Komplexität des Frontends steigt demnach drastisch an, weil jedes Formular die XML-Struktur eines PIO-ULBs kennen muss. Mit Hilfe von XML-Pfaden wird jede Nutzereingabe an der richtigen Stelle der Struktur gespeichert. Diese XML-Pfade werden im Quellcode als statische Strings hinterlegt, bei denen sich schnell Tippfehler einschleichen. In diesem Fehlerfall würde das Backend einfach einen falschen XML-Pfad erzeugen, obwohl dieser laut PIO-ULB-Spezifikation nicht valide ist. Um diese Tippfehler während der Entwicklungsphase zuverlässig zu identifizieren, wurde im Backend das Modul *„PathValidator“* (siehe Abb. 17) implementiert.

Die Alternative wäre ein statisches Backend, welches die XML-Struktur eines PIO-ULBs kennt. Das Frontend müsste also keine XML-Pfade für jedes Eingabefeld bereithalten, sondern könnte einfach Backend-Funktionen wie „createContactPerson(...)“ aufrufen, um eine neue Kontaktperson-Ressource anzulegen. In diesem Fall muss das Frontend keine XML-Strukturinformationen bereithalten. Die Anwendung eines statischen Backends hätte weiterhin die Verwendung von bereits bestehenden FHIR Implementierungen ermöglicht und den Programmieraufwand reduziert. Im Rahmen des PIO-ULB Editors ist ein generisches Backend unpraktikabel, es verursacht aber keine schwerwiegenden technischen Schulden. Ein generisches Backend ist erst sinnvoll, wenn ein Editor für unterschiedliche MIO- und PIO-Typen entwickelt werden soll.

Die Datentypkonformität zu FHIR wird durch die **PrimitiveDataTypes** gewährleistet. In der aktuellen Ausbaustufe des PIO-ULB Editors bringt dies wenig Vorteile, ist jedoch mit erheblichem Mehraufwand verbunden. Der Mehraufwand wird erst beim Import eines PIO-ULBs erkennbar. Da alle Daten in der XML-Datei als String vorliegen und FHIR keine Möglichkeit bietet, Datentypinformationen im XML-Dokument zu speichern, müssen alle Daten aufwendig ihn den richtigen PrimitiveDataType konvertiert werden. Der Import einer XML-Datei wird im Backend ausgeführt, das Backend verfügt jedoch aufgrund seiner generischen Architektur über keine XML-Strukturinformationen und somit über keine Datentypinformationen. Aus diesem Grund muss dem Backend die ResourceLookUpTable (siehe Tab. 2) aus dem FHIR-Repository bereitgestellt werden. Diese Look-Up-Table enthält alle validen XML-Pfade eines PIO-ULBs und dessen Datentypen, wodurch gegebenenfalls etwas Typsicherheit bei der Implementierung gewonnen wird. Rückblickend hätten alle primitiven Daten (egal ob String, Code oder Integer) als String repräsentiert werden können, da die Daten in der XML-Datei sowieso als String hinterlegt sind. Dadurch wäre die aufwendige Look-Up-Table überflüssig geworden.

Die rekursive Implementierung der **SubTrees** erzeugt einen hohen Konvertierungsaufwand bei jeder HTTP-Kommunikation zwischen Front- und Backend. Wird beispielsweise eine Allergie-Ressource als SubTree vom Backend an das Frontend geschickt, muss zuerst die Backend-Datenstruktur (*„entryType“*, siehe Abb. 19) in einen rekursiven SubTree konvertiert werden. Weil Daten, die über HTTP-Requests verschickt werden, als JSON übermittelt werden und somit Datentypen verloren gehen, müssen vor dem Versenden Informationen über die verwendeten primitiven Datentypen an jeden einzelnen SubTree angehängt werden. Das Frontend verwendet diese Informationen, um wieder Objekte des richtigen primitiven Datentyps zu erzeugen. Das Anhängen von Datentypinformationen und das Rekonstruieren der PrimitiveDataTypes muss für beide Kommunikationswege (vom Frontend zum Backend und umgekehrt) funktionieren, weshalb die entsprechende Logik im Front- und Backend benötigt und vom Meta-Repository bereitgestellt wird. Der erläuterte Konvertierungsaufwand erzeugt keinen merklichen Performanceverlust zur Programmlaufzeit, jedoch hätte die direkte Verwendung der Backend-Datenstruktur (*„entryType“*, siehe Abb. 19) anstelle der rekursiven SubTrees den Quellcode einfacher und schlanker gestaltet.

Das implementierte **Login-Verhalten** des PIO-ULB Editors erfüllt seinen Zweck im Rahmen der Prototypentwicklung, ist jedoch fehleranfällig. Es ist ein gewolltes Softwareverhalten der *„localVersion“*, dass sich der Nutzer mit seinem Namen erneut in die zuvor erstellte Session einloggen kann. Feldbeobachtungen bei Patientenüberleitungen ergaben, dass das Pflegepersonal während der Bearbeitung eines ULBs mehrfach Unterbrechungen (Telefonanrufe, Kollegen, Notfälle) ausgesetzt war und den Büroarbeitsplatz bzw. den Computer verlassen musste. Die Bearbeitung des ULBs wurde zu einem späteren Zeitpunkt fortgesetzt. Um einen Datenverlust während der Abwesenheit zu vermeiden, bleiben Sessions längere Zeit im Backend gespeichert. Sollte der Browser-Tab während der Abwesenheit geschlossen worden sein (z.B. versehentlich durch einen Kollegen), kann der Nutzer durch einen erneuten Login mit demselben Namen die Daten wieder laden. Melden sich zwei Nutzer zur selben Zeit mit demselben Namen an (bei Verwendung der *„localVersion“*), arbeiten beide Nutzer auf derselben Datenbasis. Dadurch stürzt das Programm nicht ab, Daten die ein Nutzer im Editor einträgt erscheinen jedoch automatisch beim anderen Nutzer und umgekehrt. Dieser Nebeneffekt wird im Rahmen der Prototypentwicklung toleriert.

Die *„webVersion“* erstellt für jeden Frontend-Client einen einmaligen *„fingerPrint“*, welcher oben beschriebenen Fehlerfall verhindert. Dadurch ist ein erneuter Login mit demselben Namen nicht mehr möglich.

Die PIO-ULB übergreifende Speicherung von FHIR-Ressourcen im **Adressbuch** des PIO-ULB Editors verursacht Identifizierungsprobleme. Aktuell funktioniert die Identifizierung von Adressbucheinträgen anhand der UUID, welche der FHIR-Ressource innerhalb des PIO-ULBs zugeordnet wurde. Diese UUID stellt eine eindeutige Identifizierbarkeit der Ressource innerhalb eines FHIR-Bundles (= PIO-ULB) sicher. Ihr Gültigkeitsbereich beschränkt sich also auf einen speziellen PIO-ULB. Wird diese UUID nun in einem übergeordneten Adressbuch zur Identifizierung verwendet, kann es zu Problemen kommen, wie folgendes Beispiel zeigt: Einrichtung A speichert ein örtliches Krankenhaus im

Adressbuch ab. Die UUID für die Organisationsressource wird automatisch generiert. Erhält nun Einrichtung A einen PIO-ULB von Einrichtung B, werden bei einem Import alle Organisationsressourcen mit dem Adressbuch synchronisiert. Die Synchronisation ist nötig, da alle FHIR-Organisationsressourcen im Editor aus dem Adressbuch geladen werden. Hat nun Einrichtung B dasselbe örtliche Krankenhaus mit einer anderen UUID versehen, wird für das Krankenhaus ein neuer Adressbucheintrag angelegt. Folglich existieren jetzt zwei Einträge für dasselbe Krankenhaus mit unterschiedlichen UUIDs. In einem anderen Fall könnte Einrichtung B eine UUID verwenden, die im Adressbuch von Einrichtung A bereits existiert. In diesem Szenario wird die Organisationsressource aus dem PIO-ULB von Einrichtung B nicht mit dem Adressbuch synchronisiert, weil die UUID schon vergeben ist. Wird der importierte PIO-ULB nun in der Editoransicht visualisiert, lädt der PIO-ULB Editor die falsche Organisationsressource aus dem Adressbuch.

Ähnliche Probleme werden auf Softwarehersteller zukommen, die künftig MIOs bzw. PIOs implementieren und Adressbuchfunktionalitäten umsetzen wollen. Eine bundesweite Lösung, um das Identifizierungsproblem zu adressieren, könnte der Aufbau einer zentralen Datenbank für Organisationen des Gesundheitswesens sein, die sich um eine eindeutige Identifizierung der Ressourcen kümmert.

12. Glossar

Hier werden wichtige Domänenbegriffe und technische Begriffe, die Stakeholder kennen sollten, wenn sie über die Architektur des Systems diskutieren, aufgelistet und erklärt.

Begriff	Erklärung
Überleitungsbogen	Ein Überleitungsbogen ist ein Dokument, das wichtige medizinische und pflegerische Informationen beim Wechsel eines Patienten zwischen verschiedenen Versorgungseinrichtungen übermittelt. Er unterstützt eine lückenlose und sichere Weiterversorgung, etwa beim Übergang vom Krankenhaus in die Pflege.
MIO / PIO	MIOs (Medizinische Informationsobjekte) und PIOs (Pflegerische Informationsobjekte) sind standardisierte Datensätze (= XML) zur strukturierten Erfassung und Darstellung medizinischer bzw. pflegerischer Informationen. Sie fördern die Interoperabilität und den einheitlichen Austausch von Gesundheitsdaten im digitalen Gesundheitswesen und sind FHIR konform.
PIO-ULB Editor	Die Software, die im Rahmen des Care Regio Forschungsprojektes von der Technischen Hochschule Augsburg entwickelt wurde und Gegenstand dieser Softwaredokumentation ist. Der PIO-ULB Editor ermöglicht das Erstellen, Editieren und Generieren von PIO-ULB Dateien (= XML-Datei).
PIO-Small	Eine Aufwand-Nutzen-Abschätzung ergab, dass der Umfang des PIO-ULB Standards für den pflegerischen Alltag zu umfangreich und komplex ist. Daraufhin wurde beschlossen, dass nur eine sorgfältig ausgewählte Teilmenge des Standards durch den PIO-ULB Editor verarbeitet werden kann. Diese Teilmenge wird als PIO-Small bezeichnet.
Telematikinfrastruktur (TI)	Die Telematikinfrastruktur (TI) ist das digitale Netzwerk des deutschen Gesundheitswesens, das alle Akteure wie Ärzte, Krankenhäuser und Apotheken sicher miteinander verbindet. Sie ermöglicht den geschützten Austausch von Gesundheitsdaten und den Zugang zu digitalen Anwendungen wie der elektronischen Patientenakte.
Elektronische Patientenakte (ePA)	Die elektronische Patientenakte (ePA) ist ein digitaler Ablageort medizinischer Informationen wie Diagnosen, Befunde und Behandlungsdaten, die für Patienten und berechtigte Ärzte zentral zugänglich ist. Sie dient dazu, die medizinische Versorgung zu verbessern, indem relevante Gesundheitsdaten schnell und sicher verfügbar sind. Die ePA ist ein Dienst der Telematikinfrastruktur.
Kommunikation im Medizinwesen (KIM)	KIM (Kommunikation im Medizinwesen) ist ein sicherer E-Mail-Dienst innerhalb der Telematikinfrastruktur, der den verschlüsselten Austausch sensibler medizinischer Daten ermöglicht. Er wird von Gesundheitsberufen genutzt, um z. B. Arztbriefe oder Befunde datenschutzkonform zu versenden.
FHIR	FHIR (Fast Healthcare Interoperability Resources) ist ein internationaler Standard für den Austausch von Gesundheitsdaten in elektronischer Form. Er ermöglicht eine strukturierte, sichere und effiziente Kommunikation zwischen verschiedenen IT-Systemen im Gesundheitswesen.

FHIR-Ressource	Eine FHIR-Ressource ist eine standardisierte, strukturierte Darstellung von medizinischen Informationen, wie Patienten, Diagnosen oder Behandlungen, die in einem FHIR-konformen System gespeichert und übertragen werden. Sie bildet die Grundlage für den Austausch und die Interoperabilität von Gesundheitsdaten in elektronischen Systemen. Jede FHIR-Ressource ist anhand einer ID eindeutig identifizierbar. Im Rahmen des PIO-ULB Editors werden UUID verwendet.
FHIR-Bundle	Ein FHIR-Bundle ist eine Sammlung von mehreren FHIR-Ressourcen, die als eine Einheit zusammengefasst werden. Es wird verwendet, um eine Gruppe von verwandten medizinischen Informationen effizient und strukturiert zu übertragen. Ein PIO-ULB ist ein FHIR-Bundle als XML-Datei.
FHIR-Composition	Eine FHIR-Composition ist eine FHIR-Ressource, die alle IDs von allen FHIR-Ressourcen, die im FHIR-Bundle vorkommen referenziert. Sie kann als Inhaltsverzeichnis verstanden werden und muss in jedem FHIR-Bundle vorhanden sein.
FHIR-Strukturdefinitionen	Eine FHIR-Strukturdefinition beschreibt die Struktur und die Datentypen einer FHIR-Ressource, einschließlich der erforderlichen Felder und deren Beziehungen. Sie dient als Bauplan für die korrekte Erstellung und Validierung von FHIR-Daten, um die Interoperabilität zwischen verschiedenen Systemen zu gewährleisten.
ValueSet	Im FHIR-Kontext ist ein Codesystem eine Sammlung von Codewerten, die bestimmte Konzepte, wie Diagnosen oder Behandlungen, repräsentieren. Ein ValueSet ist eine definierte Auswahl von Codes aus einem oder mehreren Codesystemen, die für einen spezifischen Kontext oder eine Anwendung relevant sind.
Codesystem	
SNOMED	SNOMED (Systematized Nomenclature of Medicine – Clinical Terms) ist eine international anerkannte, umfassende medizinische Terminologie zur eindeutigen Beschreibung von Krankheiten, Befunden und Therapien. Sie ermöglicht eine standardisierte und computergestützte Verarbeitung medizinischer Informationen.
Meta-Repository	Das Meta-Repository ist ein Subsystem des PIO-ULB Editors, welches Klassen, Typen und Interfaces, die im Frontend und im Backend benötigt werden, bündelt und als NPM-Package bereitstellt. (siehe Kapitel 6.1)
FHIR-Repository	Das FHIR-Repository ist ein Subsystem des PIO-ULB Editors, welches Strukturinformationen über den PIO-ULB sowie verwendete Codesysteme bereitstellt. Diese Informationen werden im Frontend und im Backen benötigt. Das FHIR-Repository wird als NPM-Package bereitgestellt. (siehe Kapitel 6.2)
Resource-LookUpTable	Die Resource-LookUpTable ist ein Teil des FHIR-Repositories. Für jede Ressource sind die Profile-URL und alle möglichen XML-Pfade, an denen Daten gespeichert werden können, hinterlegt. Für jeden XML-Pfad ist der primitive Datentyp, der an dieser Stelle erwartet wird, angegeben. Die Information welcher Datentyp an welcher Stelle in der XML-Datei zu erwarten ist, wird für die Deserialisierungslogik im Backend benötigt. Falls vorhanden, werden Codesysteme oder fixe Werte, die vom PIO-ULB Standard vorgegeben sind, für entsprechende XML-Pfade auf geführt.

React	React ist eine JavaScript-Bibliothek zur Entwicklung interaktiver Benutzeroberflächen, insbesondere für Webanwendungen. Sie ermöglicht die effiziente Erstellung von wiederverwendbaren UI-Komponenten und eine dynamische Aktualisierung der Ansicht bei Datenänderungen.
Ant Design	Ant Design ist ein UI-Framework auf Basis von React, das eine umfangreiche Sammlung vorgefertigter Komponenten für die Gestaltung moderner Webanwendungen bietet. Es unterstützt ein konsistentes Designsystem und beschleunigt die Entwicklung benutzerfreundlicher Interfaces.
Editoransicht	Die Editoransicht ist ein Screen des PIO-ULB Editors, welcher Eingabefelder für das Editieren eines PIO-ULBs bereitstellt.
Formulare	Die Editoransicht im Frontend des PIO-ULB Editors ist mit Hilfe von Ant Design Formularen implementiert worden. Jeder thematische Abschnitt ist dabei ein eigenes Formular. Die Formulare speichern selbst ihre Daten und synchronisieren sich mit dem Backend. (siehe Kapitel 9.4)
onBlur-Event	Das onBlur-Event ist ein Ereignis in JavaScript (häufig in HTML oder React genutzt), das ausgelöst wird, wenn ein Element den Fokus verliert, z. B. wenn ein Benutzer ein Eingabefeld verlässt. Es wird oft verwendet, um Eingaben zu validieren oder Änderungen zu speichern. Jedes Formular in der Editoransicht schickt die Nutzereingaben mit Hilfe des onBlur-Events an das Backend, wo die Daten in einer Session gespeichert werden.
Serialisierungslogik	Serialisierungslogik bezieht sich auf den Prozess, bei dem Daten in ein Format umgewandelt werden, das für die Speicherung oder Übertragung geeignet ist, wie z. B. in JSON oder XML. Sie sorgt dafür, dass komplexe Datenstrukturen effizient und korrekt in eine sequentielle Form gebracht werden, die später wieder rekonstruiert werden kann. Im Kontext des PIO-ULB Editors ist die Backend-Logik gemeint, welche die Nutzereingaben in eine valide XML-Datei umwandelt. Die Serialisierungslogik wird also beim Export eines PIO-ULBs aktiviert.
Deserialisierungslogik	Deserialisierungslogik ist der Prozess, bei dem serielle Daten, wie etwa JSON oder XML, zurück in ihre ursprüngliche Datenstruktur umgewandelt werden. Sie ermöglicht es, gespeicherte oder übertragene Daten korrekt zu interpretieren und in einem Programm weiterzuverarbeiten. Im Kontext des PIO-ULB Editors ist die Backend-Logik gemeint, welche eine XML-Datei ausliest und in die interne Datenstruktur des Backends umwandelt. Die Deserialisierungslogik wird also beim Import eines PIO-ULBs aktiviert.
Validator Service	Der Validator Service ist ein Subsystem des PIO-ULB Editors, welches PIO-ULB-XML-Strings entgegennimmt und gegenüber der PIO-ULB Spezifikation offline validieren kann. (siehe Kapitel 6.5)
PrimitiveDataTypes	Die 12 PrimitiveDataType Klassen werden vom Meta-Repository bereitgestellt und finden sowohl im Frontend als auch im Backend Anwendung. Sie bilden Datentypen ab, die im FHIR Standard vorgesehen sind (z.B. String, DateTime, Code, Boolean, Integer, usw.). (siehe Kapitel 9.1)

SubTrees	Die SubTree Klasse wird vom Meta-Repository bereitgestellt und dient der Kommunika tion zwischen Frontend und Backend. Werden Patientendaten zum Speichern an das Ba ckend gesendet oder Patientendaten vom Backend abgefragt, geschieht dies immer als SubTree. Ein SubTree ist ein rekursiv aufgebautes Objekt, welches sich selbst mehrfach instanziieren kann. (siehe Kapitel 9.2)
RootObject	Das Herzstück des Backends ist die „Root Object“ Klasse, welche alle Daten eines PIO ULBs speichert und die Serialisierungs- bzw. Deserialisierungslogik umfasst. Für jede Session wird eine Instanz dieser Klasse angelegt. Dies bedeutet, dass in einer Session maximal ein PIO-ULB gleichzeitig bearbeitet werden kann. (siehe Abb. 18)
PIOHeader Klasse	Die Attribute „header“ und „content“ der RootObject-Klasse halten jeweils eine Instanz einer anderen Klasse (PIOHeader & PIOContent). PIOHeader speichert alle Header-Daten eines PIO-ULBs. Als Header-Daten werden alle Informationen des FHIR-Bundles und der FHIR Composition angesehen. PIOContent speichert alle inhaltlichen Daten (= Patientendaten) eines PIO-ULBs.
PIOContent Klasse	
EntryType	Die Klasse PIOContent speichert die inhaltlichen Daten (= Patientendaten) in einer speziellen internen Datenstruktur, die durch den Typ „EntryType“ beschrieben wird. Die Serialisierungslogik wandelt den „EntryType“ in das XML-Format, während die Deserialisierungslogik einen XML-String in einen „EntryType“ umwandelt. Der „EntryType“ orientiert sich stark an der XML-Struktur. (siehe Abb. 19)
XML-Pfad	Wenn das Frontend Daten zum Speichern an das Backend sendet, muss das Frontend den exakten Speicherort mitteilen, weil das Backend generisch implementiert wurde. Dies geschieht anhand von XML-Pfaden, welche beispielhaft so aussehen könnten (siehe Kapitel 6.3): *24ed8a3b-59f7-47fd-8699-bb908de982c0.KBV_PR_MIO_ULB_Patient.address[0].postalCode*
Path Validator	Während das „Root Object“ bei einem Import selbstständig XML-Pfade detektiert, die in der PIO-ULB Spezifikation nicht vorgesehen sind, übernimmt der „Path Validator“ diese Aufgabe bei einem Export. Sollte also ein Programmierfehler im Frontend vorliegen, sodass Daten unter einem Pfad abgelegt werden, der nicht spezifikationskonform ist, wird der Export fehlschlagen und die Generierung der XML-Datei abgebrochen. Der „Path Validator“ wird vom Root-Object instanziiert. (siehe Abb. 17)
Redux Store	Der Redux Store ist ein zentraler Zustandsspeicher, das den gesamten Anwendungszustand in einer Redux-gestützten Anwendung speichert. Er ermöglicht eine vorhersehbare Verwaltung und Aktualisierung des Zustands durch Actions und Reducer. (siehe Kapitel 9.3)
„localVersion“	Der PIO-ULB Editor kann abhängig von der Umgebungsvariable *VERSION_ENV* in zwei unterschiedlichen Versionen gestartet werden. Die „localVersion“ ist für den lokalen Betrieb im Netzwerk von

„webVersion“	medizinischen und pflegerischen Einrichtungen gedacht. Die „webVersion“ ist für den Betrieb auf einer öffentlichen Website gedacht, auf der der PIO-ULB Editor zu Demonstrationszwecken gehostet wird. (siehe Tab. 4)
Docker	Docker ist eine Plattform zur Erstellung, Bereitstellung und Ausführung von Anwendungen in Containern, die eine isolierte und portable Umgebung bieten. Sie ermöglicht es, Software schnell und konsistent über verschiedene Systeme hinweg zu deployen und auszuführen.