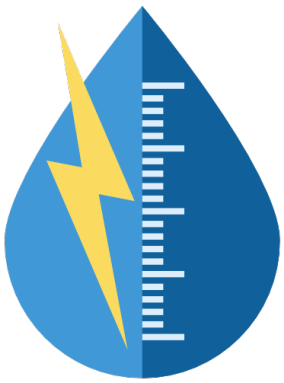
Entwurfsdokumentation
energy. by adesso
-
Softwareprojekt SoSe19
HRS3-105B



adesso energy

Massoud Vincent Shahriyari
Jan-Niklas Carstensen
Tammo Brüggemann
Richard Hanß
Christoph Fricke
Felix Rodriguez
Mattis thor Straten
Malte Clement

30. August 2019

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Dokumentaufbau

Das Ziel dieses Dokuments ist die Dokumentation der Entwicklung des Onlineportals und der App von adesso energy. Die enthaltenen Diagramme sollen neuen Teammitgliedern einen erleichterten Einstieg in die Struktur des Projekts ermöglichen und somit den Einarbeitungsprozess vor der aktiven Teilnahme an der Entwicklung verkürzen und vereinfachen. Außerdem können sich Mitarbeiter, die bereits am Projekt arbeiten, mit Hilfe dieser Entwurfsdokumentation einen Überblick über ihren aktuellen Stand und bei Unklarheiten ebenfalls über die Struktur des Projektes verschaffen. Die frühe Planung des Zusammenspiels der einzelnen Anwendungsbereiche soll zu einem reibungslosen Ablauf in der tatsächlichen Entwicklung des Onlineportals führen und ermöglicht außerdem rechtzeitige Absprachen über die Funktionalität der Schnittstellen.

Das Dokument besteht aus 8 Kapiteln, die jeweilen Kapitel werden im Folgenden kurz beschrieben. In Kapitel 2 präsentieren wir die Aufteilung unseres Teams in Gruppen, welche sich auf die Entwicklung der verschiedenen Anwendungsbereiche spezialisiert haben. In Kapitel 3 werden die REST-API Endpunkte beschrieben, sowie die verschiedenen DTO Datentypen erläutert. Daraufhin stellen wir in Kapitel 4 anwendungsbereichsspezifische Komponentendiagramme vor, welche die grobe Struktur der Anwendungsbereiche klären. In Kapitel 5 folgt ein bereichsübergreifendes Verteilungsdiagramm, welches die Verteilung der Komponenten auf die Hardware erläutert. Im 6. Kapitel befindet sich eine Sammlung von Klassendiagrammen, die wiederum die Struktur des Entwurfes der jeweiligen Anwendungsbereiche spezifizieren. Anschließend werden in Kapitel 7 die wichtigsten Abläufe von Methodenaufrufe aus den Klassendiagrammen in Sequenzdiagrammen dargestellt, um ein besseres Verständnis des Zusammenspiels der Klassen zu schaffen. Beim letzte Kapitel (8) handelt es sich um das Glossar. In diesem können die Definitionen von Grundbegriffen des Projektes nachgeschlagen werden, welche in dieser Entwurfsdokumentation ohne Erläuterung verwendet werden.

1.2 Zweckbestimmung

In diesem Kapitel wird der Mehrwert dieses Projektes verdeutlicht. Für die Abrechnungen der Gas-,Wasser- und Stromkosten müssen Energieanbieter regelmäßig die Zählerstände ihrer Kunden einfordern. Bisher war es Praxis diese per Post zu übermitteln oder online einzutragen. Wenn bei ersterem Fehler auftraten, musste weiterer Briefverkehr erfolgen.
Der Zweck des Systems ist die Vereinfachung des beschriebenen Szenarios. Der Upload von Bildern durch die App ersetzt den Briefverkehr und spart den Arbeitsaufwand für das Bearbeiten der Post ein. Administratoren können über das Back-End, an das die Kunden ihre Daten schicken, auf die Kundendaten zugreifen. Dies vermeidet Redundanzen und Fehler beim Übertragen der Daten in die Datenbank.
Aus Kundensicht ist die Benutzung komfortabel und einfach, da die App schlicht und funktional ist. Das Hochladen der Bilder ist einfach und erfordert nur geringe technische Kenntnisse. Außerdem ersetzen die Push-Notifications die, ansonsten per Brief geschickten, Aufforderungen den Zählerstand erneut abzulesen.

1.3 Entwicklungsumgebung

Software	Version	URL
Java Development Kit	8u144	`http://www.oracle.com/technetwork/java/javase/downloads/index.html`
Node	12.9.0	`nodejs.org/en/download/current`
npm	6.10.2	`npmjs.com`
React	16.9	`reactjs.org`
Create React App	3.1.1	`create-react-app.dev`
Spring	5.1.6	`projects.spring.io/spring-framework`
Docker	19.03.1	`www.docker.com`
Android SDK API	LVL 23	`https://developer.android.com/studio/releases/platforms#\6.0`
PostgreSQL	11.4	`www.postgresql.org`
Android Studio	3.5	`https://developer.android.com/studio`
Storybook	5.1	`https://storybook.js.org`

Tabelle 1.1: Enwicklungsumgebung

Kapitel 2

Team-Aufteilung

Die folgende Tabelle beinhaltet die Zuständigkeitsbereiche der Gruppe. Da das Backend besonders stark verteten ist, kann, sollte dies nötig werden, einer der Entwickler zu einem der Frontends wechseln.
Außerdem kann sich ein Entwickler verstärkt auf die Tests konzentrieren, falls dafür Bedarf entsteht.

Name	**Zuständigkeit**
Vincent	Verwaltung, Back-End u. CI/CD
Malte	Back-End
Mattis	Back-End
Felix	Back-End
Richard	App
Niklas	App
Christoph	Website
Tammo	Website

Kapitel 3

REST-API Endpunkte

Anmerkung:
Endpunkte, welche ein Paging<T> Objekt zurück geben sind pageable und sortierbar. Dafür können in dem Request weitere Parameter angeben werden. Allerdings sind wir uns noch nicht sicher, wie diese Parameter, in den von Spring angebotenen Libraries, aussehen. Deshalb haben wir diese Parameter vorerst noch nicht aufgeführt. Außerdem sind Status-Codes als Return-Wert möglich, um Fehler zu behandeln. Zum Beispiel ist an allen Endpunkten eine Authentifizierung nötig, sodass 401 ein weiterer möglicher Return-Code ist. Alternative Fehler-Codes haben wir vorerst aber noch nicht genau definiert.

```
interface Paging<T> {
        content: T[]
        size: number
        page: number
        totalPages: number
        total: number
        first: boolean
        last: boolean
}

interface MeterDTO {
        id: string
        type: "water" | "eletricity" | "gas"
        name: string
        ownerId: string | null
        lastReading: ReadingDTO
        meterNumber: string
        createdAt: UTC string
        updatedAt: UTC string | null
        deletedAt: UTC string | null
}
```

```
interface ReadingDTO {
        id: string
        meterId: string
        ownerId: string
        value: string
        trend: number
        lastEditorName: string
        lastEditReason: string
        createdAt: UTC string
        updatedAt: UTC string | null
        deletedAt: UTC string | null
}

interface IssueDTO {
        id: string
        email: string
        name: string
        subject: string
        message: string
        status: "UNRESOLVED" | "RESOLVED"
        createdAt: UTC string
        updatedAt: UTC string | null
        deletedAt: UTC string | null
}

interface UserDTO {
        id: string
        customerId: string
        firstName: string
        lastName: string
        email: string
        createdAt: UTC string
        updatedAt: UTC string | null
        deletedAt: UTC string | null
}

interface PictureResultDTO {
        meterId: string
        value: string
}
```

Zähler

GET /api/meters	Liste aller Zähler	200 : Paging <MeterDTO>
POST /api/meters	Zähler erstellen	201 : MeterDTO
PUT /api/meters/{mid}	Zähler updaten	200 : MeterDTO
DELETE /api/meters/{mid}	Zähler löschen	204

Zählerstände

GET /api/meters/{mid} /readings	Alle Zählerstände zu einem Zähler	200 : Paging <UserDTO >
POST /api/meters/{mid} /readings	Neuen Zählerstand einem Zähler hinzufügen	201 : ReadingDTO
PUT /api/meters/{mid} /readings/{rid}	Einen Zählerstand eines Zählers updaten	200 : ReadingDTO
DELETE /api/meters/{mid} /readings/{rid}	Einen Zählerstand eines Zählers entfernen	204
POST /api/picutre	Zählerstand + Zähler vom Bild erkennen	200 : PictureResultDTO

Benutzer

GET /api/users	Liste aller Benutzer	200 : Paging <UserDTO >
GET /api/users/{uid}	Infos zu einem Benutzer	200 : UserDTO
GET /api/users/{uid}/meters	Liste aller Zähler zu einem Benutzer	201 : Paging <MeterDTO >
GET /api/users/{uid}/meters/{mid} /readings	Liste aller Zählerstände für einen Zähler von einem Benutzer	200 : Paging <ReadingDTO >
POST /api/users	Benutzer hinzufügen	201 : UserDTO
PUT /api/users/{uid}	Benutzer updaten	200 : UserDTO
DELETE /api/users/{uid}	Benutzer löschen	204

Ticket

GET /api/issues	Liste aller Tickets	200 : Paging <IssueDTO >
POST /api/issues/{iid}	Ticket hinzufügen	201 : IssueDTO
PUT /api/issues/{iid}	Updaten eines Tickets	200 : IssueDTO
DELETE /api/issues/{iid}	Ticket löschen	204

Authentifizierung

POST /api/login	Login Token für Benutzer erhalten	200 : JWT Token
GET /api/logout	Benutzer ausloggen	204

Kapitel 4

Komponentendiagramme

4.1 Website

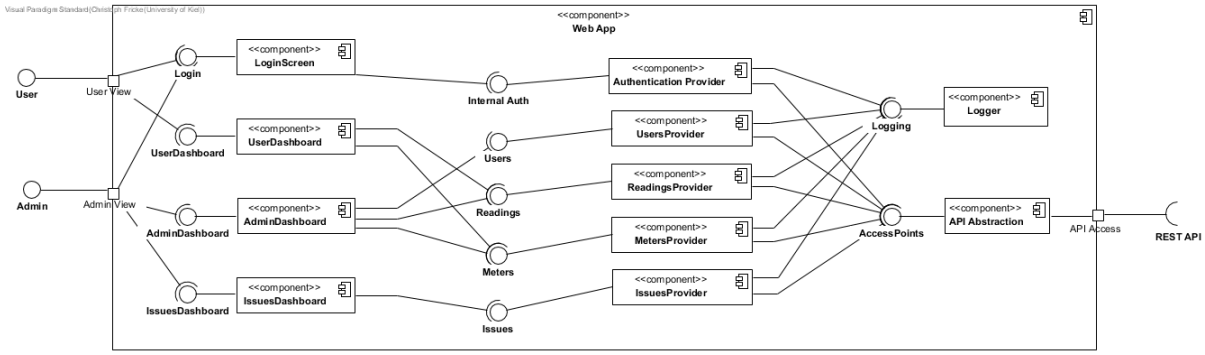


Abbildung 4.1: Komponentendiagramm - Website

In der Architektur der Web-Application kann man grob in 3 Spalten unterteilen.

Hierbei stellt die rechte Spalte die Service Ebene dar, in ihr befindet sich eine API Abstraction. Die API Abstraction abstrahiert alle REST-Endpunkte, welche wir in der Web-Applikation benötigen. Außerdem liegt hier ein Logger, welcher uns im Entwicklungsprozess Fehlermeldungen und Logstatements auf der Console ausgibt. Optional könnte man diese auch an einen Logservice (z.B.: Sentry) schicken.

In der mittleren Spalte befinden sich alle Provider. Diese stellen der Benutzeroberfläche globale Informationen (state) zur Verfügung. Hier werden API-Anfragen, welche vom Service abstrahiert sind, getriggert und die entsprechenden Daten gesammelt und gecached.

In der linken Spalte liegen die Benutzeroberflächen, auf die Benutzer und Administratoren zugreifen. Diese benutzen die Provider, um Informationen von der API zu erhalten.

4.2 App

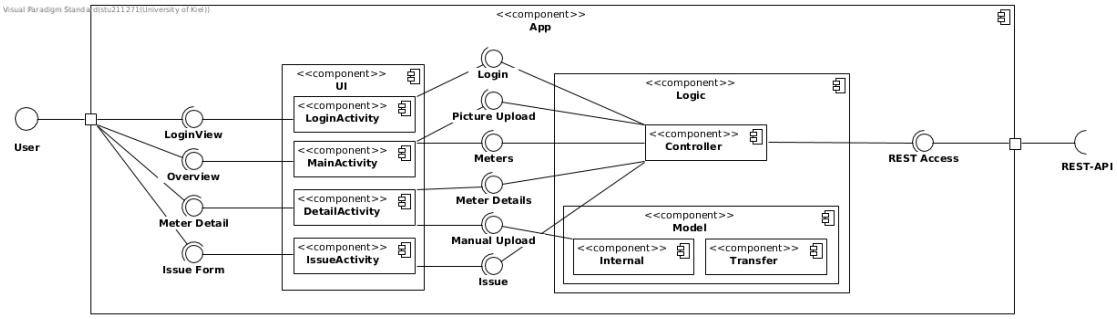


Abbildung 4.2: Komponentendiagramm - App

Die App ist in mehrere Komponenten aufgeteilt, von denen die meisten direkt die Java-Packages des Projektes darstellen. Ausnahmen sind die Actvities in der UI Komponente, die hier eingetragen werden, obwohl sie häufig nur eine "Activity"-Klasse umfassen, weil sie darüber hinaus XML Dateien und andere Resourcen nutzen.
Die meisten Anfragen an die UI gehen an die Controller Klassen, die als Fassade zwischen Front- und Backend der App dienen. Bei manuellen Uploads kommunuiziert die UI direkt mit der "Meter"-Klasse, die in der "Internal"-Komponente liegt.

Die Transfer Komponente enthält alle DTOs des Projekts, und stellt paging support bereit, der von der Meter-Klasse in der "Internal"-Komponente benutzt wird.
Internal enthält Datenobjekte, die die DTOs für den Rest der Anwendung abstrahieren.

4.3 Back-End

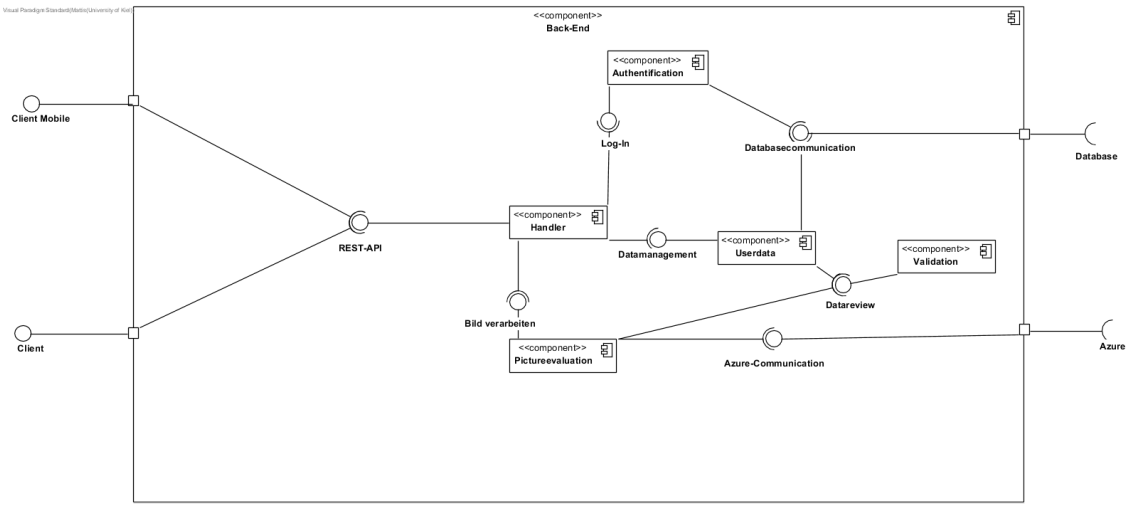


Abbildung 4.3: Komponentendiagramm - Back-End

Zusammengesetzt ist das Back-End aus mehreren Komponenten und Schnittstellen, die untereinander interagieren.
Dabei regelt die "Handler"-Komponente den Dateneingang durch Nutzeranfragen, indem sie die Daten an die für sie zuständigen Subkomponenten weiterleitet. Dies geschieht über die von diesen Komponenten angebotenen Schnittstellen. Die Nutzeranfragen aus App und Browser werden dabei durch die "REST-API"-Schnittstelle angenommen.
Um zu überprüfen, ob Nutzer privilegiert sind bestimmte Anfragen zu stellen, existiert die "Authentification"-Komponente. Diese verhindert damit, dass Nutzer Operationen durchführen, zu welchen sie nicht befugt sind. Zu diesem Zweck stellt sie dem "Handler" eine "Verify"-Schnittstelle zur Verfügung. Um die Befugnis eines Benutzers für eine bestimmte Operation zu überprüfen, benutzt die "Authentifikation" die "Database-Communication"-Schnittstelle. Über diese werden die benötigten Rechte mit den in der Datenbank gespeicherten Rechten des Benutzers abgeglichen.
Weiterhin wird diese "Data-Communication"-Schnittstelle von der "User-Data"-Komponente genutzt. "User-Data" ermöglicht bei entsprechenden Privilegien das Abrufen oder Ändern von Kundendaten. Diese Komponente wird insbesondere genutzt, um Zählerstände zu aktualisieren.
Die gewünschten Zugriffe auf Kundendaten erfolgen über die von der "User-Data"-Komponente bereitgestellte "Data-Management"-Schnittstelle.
Bei Veränderungen der Daten in "User-Data" werden diese durch die "Datareview"-Schnittstelle an die "Validation"-Komponente weitergeleitet. Dort werden Zählerstände und Nummern auf ihr Format, sowie Passwörter auf Sicherheitsstandards, überprüft.
Auf die "Data-Review"-Schnittstelle greift ebenfalls die "Picture-Evaluator"-Komponente zu, um Zählernummern und Zählerstände auf Validität zu überprüfen. Diese Werte werden generiert von der "adesso-Picture-Recognition", welche die Bilder extern auswertet. Verbunden sind diese dabei über die "Picture-Evaluation"-Schnittstelle. Die Bilder, welche ausgewertet werden sollen, gelangen über die "Picture-Handling"-Schnittstelle, welche von der "Picture-Evaluator"-Komponente angeboten wird, in eben diese. Umgekehrt werden die daraus erhaltenen Daten über die gleiche Schnittstelle an die "Handler"-Komponente zurückgesendet. Die "Picture-Evaluator"-Komponente fungiert somit als Knotenpunkt für die Bildverarbeitung.

Kapitel 5

Verteilungsdiagramm

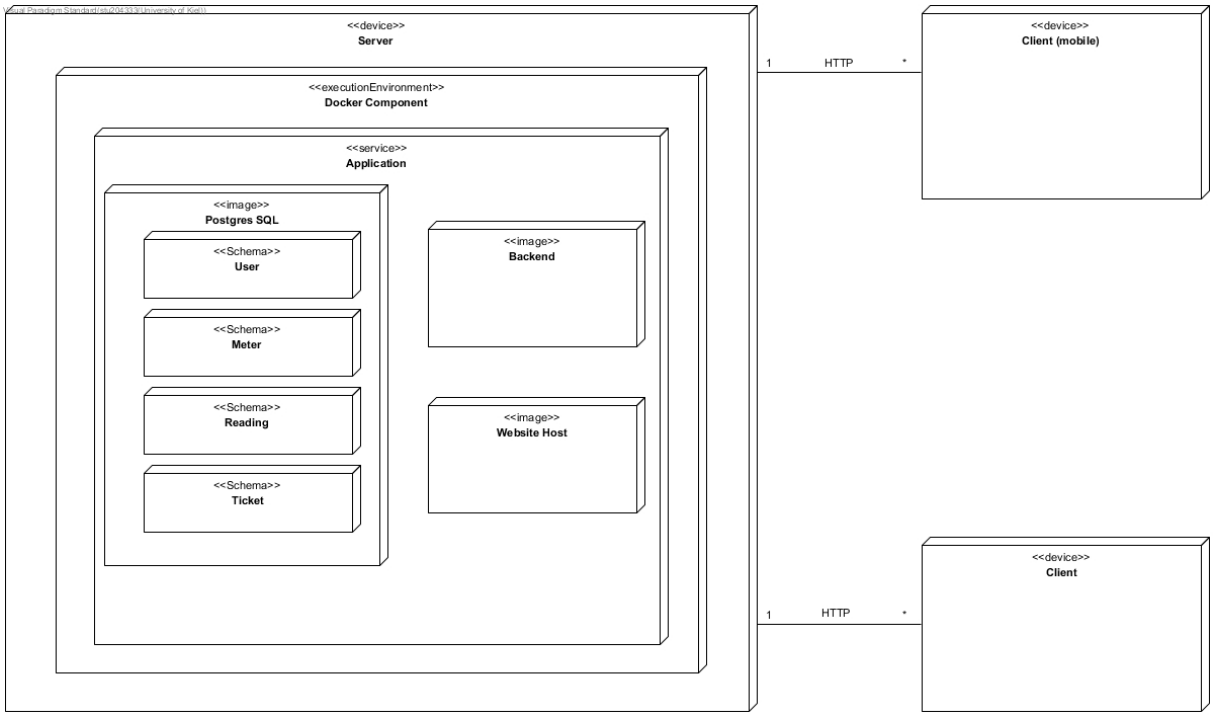


Abbildung 5.1: Verteilungsdiagramm

Im Allgemeinen läuft die gesamte Software auf dem selben Server. Durch Docker werden einzelne Komponenten aber auf unterschiedliche Container aufgeteilt.
So läuft in einem Container die Software für das Backend, die über eine REST-API mit dem Frontend kommuniziert. Die Web und App Anwendungen können dann HTTP Anfragen an das Backend stellen.
Ein weiterer Container beinhaltet die SQL-Datenbank. Diese beinhaltet Schemata für Nutzer, Zähler, Zählerstände und Tickets.

Kapitel 6

Klassendiagramme

6.1 Web-App

6.1.1 Einleitung zu den Klassendiagrammen

In der Web-Application benutzen wir Typescript, um in Javascript Typsicherheit zu gewähren. In den Diagrammen werden deshalb Typescript-Spezifische Typen und Utility-Typen benutzt. Hierfür siehe Typescript Docs für Advanced Types und Utility Types:
`http://www.typescriptlang.org/docs/handbook/advanced-types.html`
`http://www.typescriptlang.org/docs/handbook/utility-types.html`

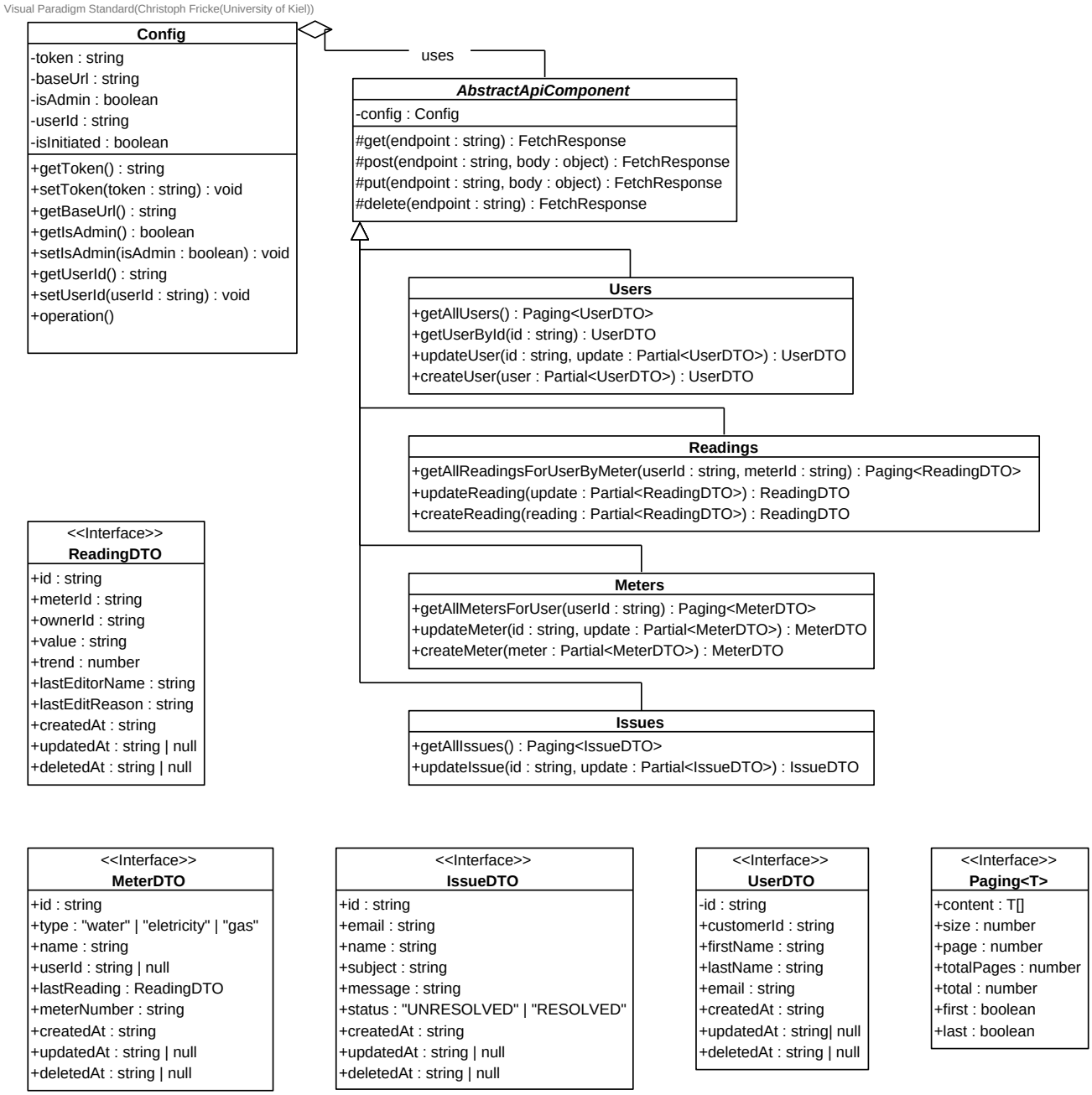


Abbildung 6.1: Klassendiagramm - API Abstraction

Beschreibung zum obigen Klassendiagramm - API Abstraction

Die verschiedenen DTO-Interfaces sind Datentypen, welche über die REST-Endpunkte verschickt werden. Sie dienen als Datentyp für Javascript Objekte in Typescript. In der API-Abstraktion halten wir eine eigene Config-Klasse, damit Komponenten, welche die Abstraktion benutzen, nicht bei jedem Aufruf den Token und die Base-URL mitgeben müssen. Dadurch muss die Config nur einmal initialisiert werden, sobald der Benutzer die Website lädt.

Eine abstrakte Klasse stellt konfigurierte Netzwerkfunktionen für die spezialisierten Klassen zur Verfügung, sodass die Methode zum Erstellen von Netzwerkanfragen (fetch oder axios) leichter ausgetauscht werden kann.
Die spezialisierten API Components und Config sind Singletons. Eine Index-Datei im API-Package exportiert für jede Klasse ein Objekt, wodurch auf einfache Art und Weise ein Singleton-Pattern in JS realisiert werden kann. Zum Testen kann die Klasse weiterhin importiert werden, sodass z.B. für Tests neue Objekte erzeugt werden können. Dieses Verhalten wäre auch durch Mocking erreichbar, ist jedoch aufwendiger umzusetzen und erzeugt mehr Overhead.

Klassenname	Aufgabe
Config	Speichert API Access Informationen wie Token und Base-URL. Auf diese kann nicht zugegriffen werden, bevor die Config initialisiert wurde.
AbstractApiComponent	Stellt konfigurierte Netzwerkfunktionen für die spezialisierten Klassen zur Verfügung.
Users	Beinhaltet Funktionen, um auf die REST-Endpunkte zuzugreifen, welche inhaltlich zu den Benutzern gehören.
Readings	Beinhaltet Funktionen, um auf die REST-Endpunkte zuzugreifen, welche inhaltlich zu den Zählerständen gehören.
Meters	Beinhaltet Funktionen, um auf die REST-Endpunkte zuzugreifen, welche inhaltlich zu den Zählern gehören.
Issues	Beinhaltet Funktionen, um auf die REST-Endpunkte zuzugreifen, welche inhaltlich zu den Tickets gehören.

Tabelle 6.1: Klassenbeschreibung - API Abstraction

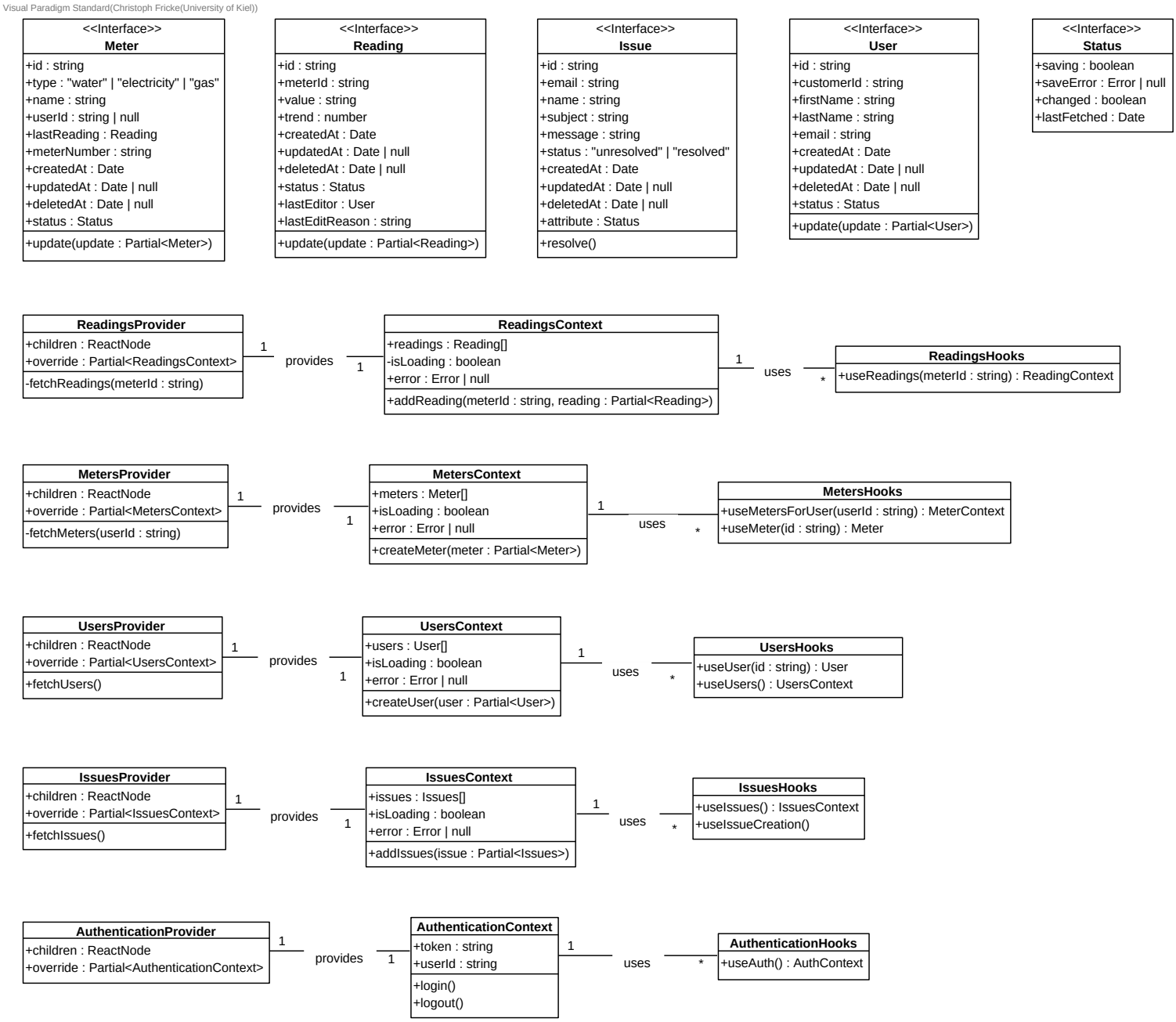


Abbildung 6.2: Klassendiagramm - Provider

Beschreibung zum obigen Klassendiagramm - Provider

Für das Klassendiagramm der Provider ist es wichtig, React Context zu verstehen. Der Override in den Providern erlaubt es uns, den Context in Tests auszutauschen. Dadurch können Components, deren Kinder oder diese selber einen Access-Hook benutzen, getestet werden, ohne dass der Hook gemockt werden muss. Hooks sind eine React-Funktionalität, um Logik innerhalb von Components zu abstrahieren und wiederzuverwenden. Ein Context wird hier als Klasse dargestellt. In Wahrheit ist es aber ein Objekt, welches von React verwaltet wird. Dadurch werden konsumierende Components bei Updates automatisch neu gerendert. Das Ganze ist vergleichbar mit einem Observer Pattern.
Die Hook-Klassen existieren nur im Diagramm. Im Quellcode sind es nur die aufgelisteten Hooks, welche von den UI Components benutzt werden können, um Zugriff auf die Contexts zu bekommen. Die Hooks dienen damit als Fassade, sodass die React Contexts auch durch Redux für das globale State Management ersetzt werden könnten. Provider Components bringen die verschiedene Contexts in den React-Tree, damit diese für die Hooks in den UI Components zur Verfügung stehen. Die Interfaces, welche im Context gehalten werden, sind erweiterte DTO-Interfaces, welche zusätzliche interne Informationen speichern.

Klassenname	Aufgabe
Context	Contexts stellen ihre Informationen global zur Verfügung.
Provider	Die Provider rendern ihren zugehörigen Context in den React-Tree.
Hooks	Hooks sind Zugriffsfunktionen, welche als Fassade für die zugehörigen Contexts dienen, damit die Components nicht direkt auf den Context zugreifen.

Tabelle 6.2: Klassenbeschreibung - Provider

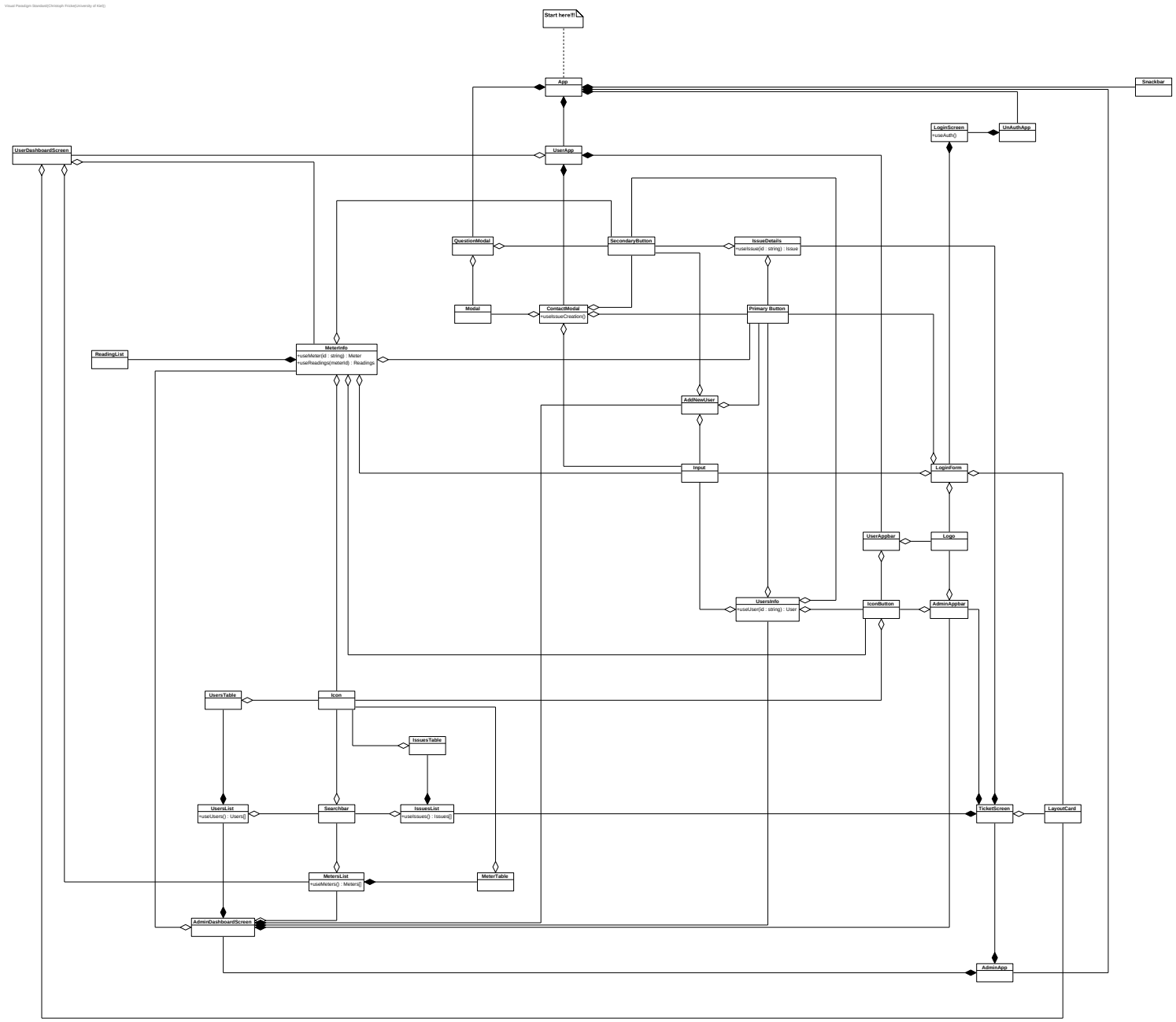


Abbildung 6.3: Klassendiagramm - Benutzeroberfläche Components

Beschreibung zum obigen Klassendiagramm - Benutzeroberfläche Components

Alle Klassen aus dem Diagramm sind React-Components. Damit es übersichtlicher ist, haben wir die Properties und den internen State, sowie die Kardinalitäten der Assoziationen, rausgelassen und nur Hooks aus der Provider-Schicht hinzugefügt wo wir vermuten auf den globalen State zuzugreifen.

Allgemein bestehen die Benutzeroberflächen aus Kompositionen von UI Components welche von React in den HTML-DOM gerendert werden.

Klassenname	Aufgabe
App	Die App ist die Hauptkomponente, welche in den HTML-DOM gerendert wird. Basierend auf dem Status des Benutzers rendert diese entweder die User-App, die Admin-App oder die Unauth-App, welche den Login für nicht eingeloggte User beinhaltet.
Layout Card	Die Layout Card rendert eine weiße Box welche als Container für ihre inneren Components dient.
Snackbar	Die Snackbar sind kleine Pop-Up-Benachrichtigungen, welche dem Benutzer Statusmeldungen mitteilen.
UsersList	Die UsersList rendert eine Liste von allen Benutzern mit einer Suchbar .
MetersList	Die MetersList rendert eine Liste von allen Zählern eines Benutzers mit einer Suchbar.
IssuesList	Die IssuesList rendert eine Liste von allen Tickets mit einer Suchbar .
UsersInfo	Enthält Informationen über die Benutzer und die Möglichkeit diese zu editieren.
IssueDetails	Enthält eine detaillierte Ansicht über ein Ticket und Interaktionsmöglichkeiten.
MeterInfo	Enthält Informationen über einen Zähler und seine Zählerstände.

Tabelle 6.3: Klassenbeschreibung - Benutzeroberfläche Components

6.2 Klassendiagramme - Back-End

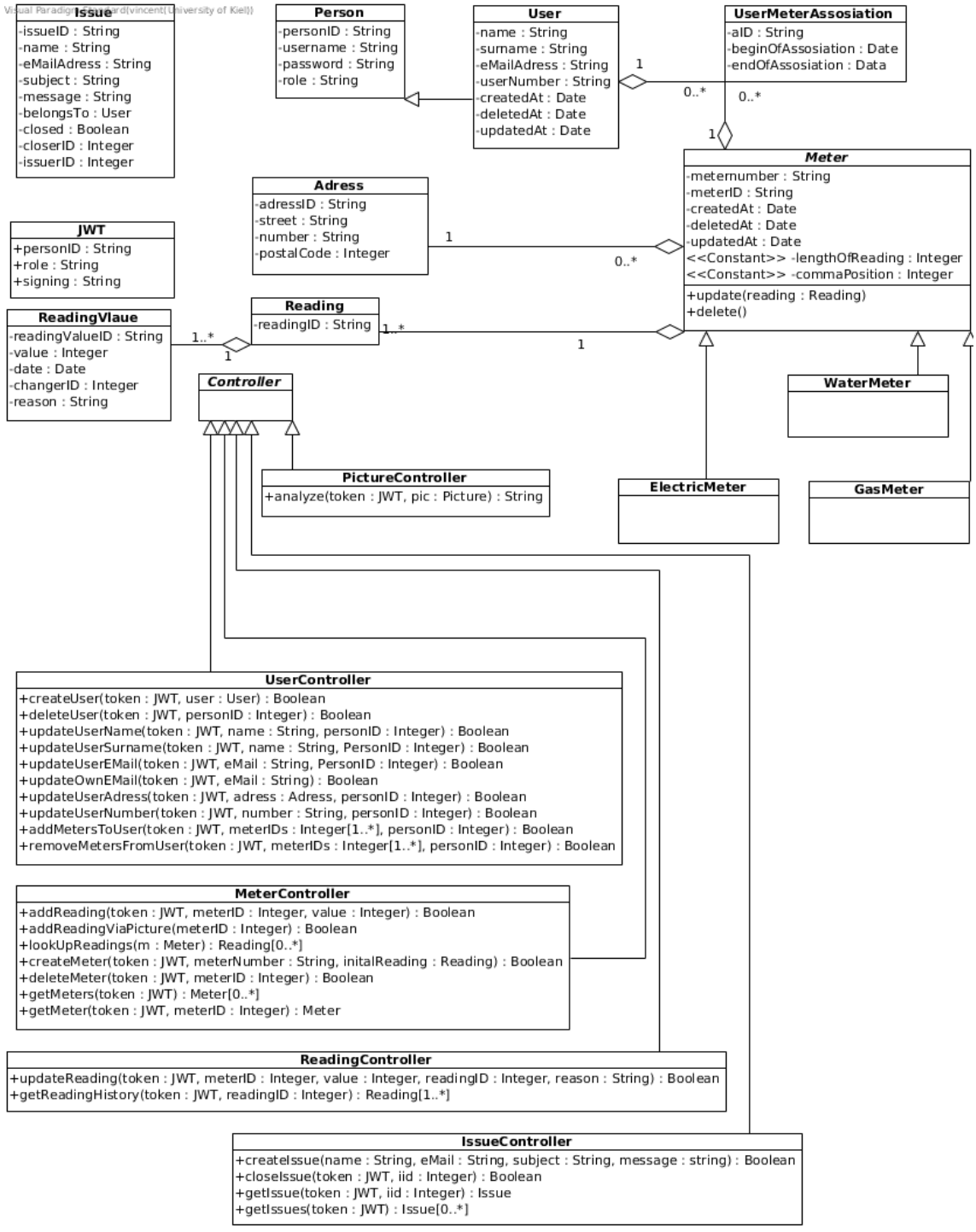


Abbildung 6.4: Klassendiagramm - Back-End

Bei allen Objekten, die Entities des Models darstellen (Person, User, UserMeterAssociation, Meter, Reading, ReadingValue, Adress, Issue), handelt es sich um Spring Entity-Objekte. Die normalerweise anfallenden Klassen (z.B. Repository-Klassen) werden zwar generiert, aber zu Übersichtszwecken nicht im Diagramm aufgeführt.
JWT steht für JSON Web Token. JWTs werden genutzt, um bei Anfragen an die REST-API User und Administratoren zu authentifizieren. Die REST-API wird durch den Controller bereitgestellt. Man kann daher auch aus jeder Anfrage, die einen JWT enthält, einen konkreten User herauslesen. Da es sich um ein JSON-Objekt handelt, wird es intern nur als String wahrgenommen, aber zur Übersicht im Diagramm wurde es mit dem geplantem Inhalt als Java-Klasse aufgeführt.
Da Kunden aus einer Wohnung ausziehen können und neue Kunden dort einziehen können, benötigen Kunden und Zähler eine Komponente oder Funktion, welche einen Zähler zu einem gewissen Zeitpunkt einem Kunden zuordnet. Zu diesem Zweck existiert die UserMeterAssociation.
Ein Zähler besitzt mindestens einen aktuellen Zählerstand und gegebenenfalls mehrere alte Zählerstände. Außerdem besitzt er eine Adresse, an der er montiert ist.
Das Attribut lengthOfReading wird von den konkreten Zählern geerbt und beschreibt, welche Länge eine Eingabe dieser Zählerart besitzt. Das Attribut commaPosition beschreibt dabei, die Anzahl der Ziffern von commaPosition, die Nachkommastellen sind.
Ein Zählerstand hat mindestens einen Wert, der beim Ablesen angegeben wird. Da er aber unter Umständen von Administratoren geändert werden kann, werden zusätzlich alle Versionen des Zählerstandes inklusive der Person, die ihn verändert hat, gespeichert. Ebenfalls wird der Grund für die Änderung gespeichert. Beim Erstellen eines Zählerstandes (Reading) wird ein neues ReadingValue bestehend aus dem Stand des Zählers sowie Default-Werten für Grund und Ersteller generiert, um klar zu machen, dass diese Daten noch nicht manipuliert wurden.
Die Controller dienen zum Auslesen und Manipulieren der Daten des Models und sind jeweils darauf spezialisiert, die Anfragen von Nutzern, Zählern oder Einträgen zu verarbeiten.

6.3 Klassendiagramme - App

Die App ist intern in eine View- und eine Model/Controller-Komponente aufgeteilt. Es folgen Klassendiagramme für beide Komponenten.

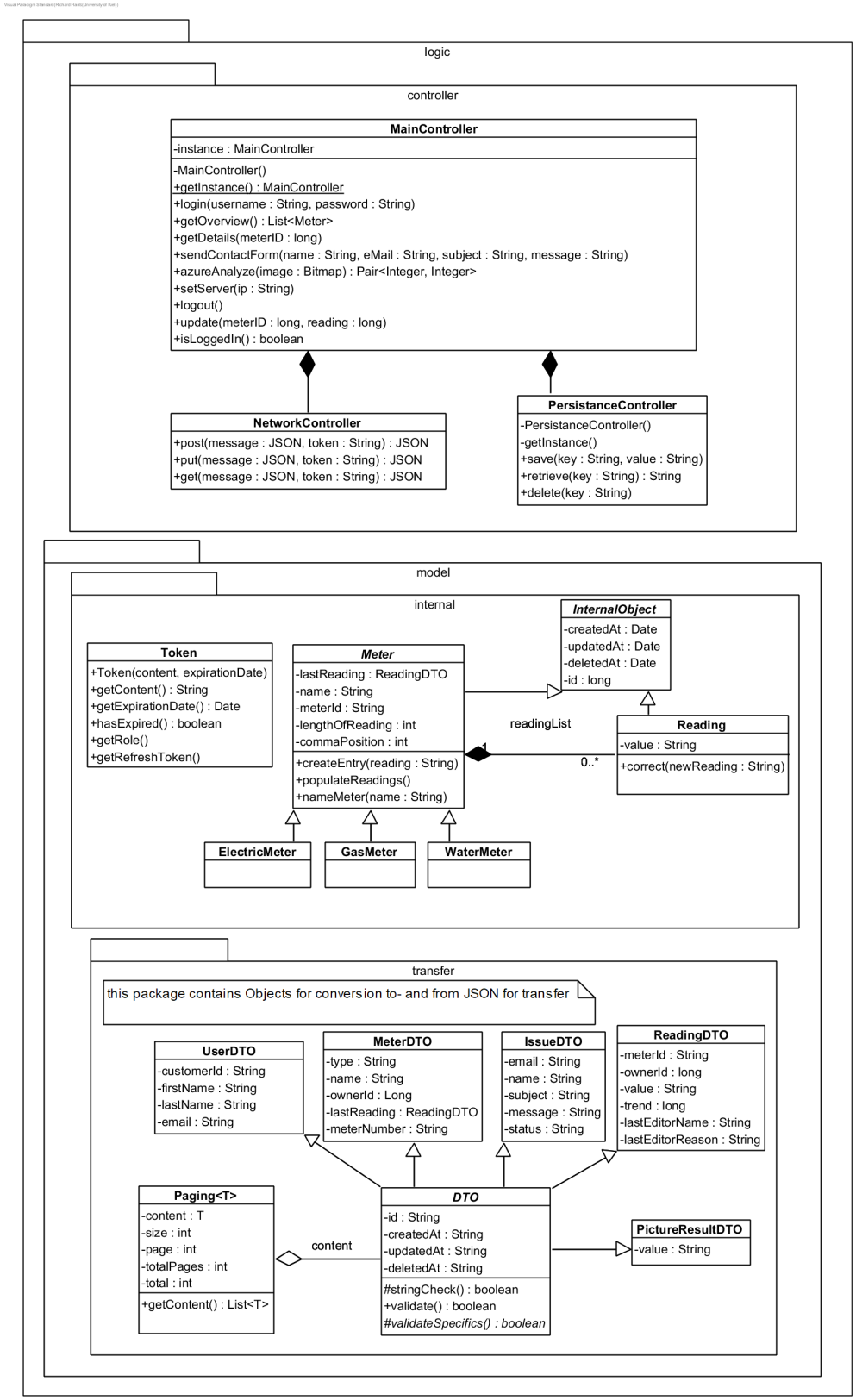


Abbildung 6.5: Klassendiagram - Logik der App

Erklärung - Model & Controller der App

Im Paket "logic" befinden sich die Pakete "model" und "controller". Klassen im Controller-Paket sind verantwortlich für das tatsächliche Ausführen der Operationen, die vom User über Interaktion mit den Views gestartet werden.
Die Singleton-Klasse MainController ist hier erster Ansprechpunkt für die Views und stellt Methoden zum Ein- und Ausloggen, sowie zum Analysieren von Bildern, Fetchen von Zählern, Senden von Kontaktanfragen und Ändern vom zu benutzendem Server bereit.
Der MainController benutzt zum Senden von HTTP-Requests den NetworkController und zum Speichern von persistenten Daten (wie z.b. OAuth Token) den PersistanceController. Das Model ist in zwei Pakete unterteilt. "internal" enthält Objekte, die den Views zur Anzeige übergeben werden, sowie eine Repräsentation des OAuth-Tokens.
Im "transfer"-Paket befinden sich Klassen, die identisch zu den über REST zu übertragenden JSON Objekten aufgebaut sind. Sie werden über eine Library von und zu JSON konvertiert werden.

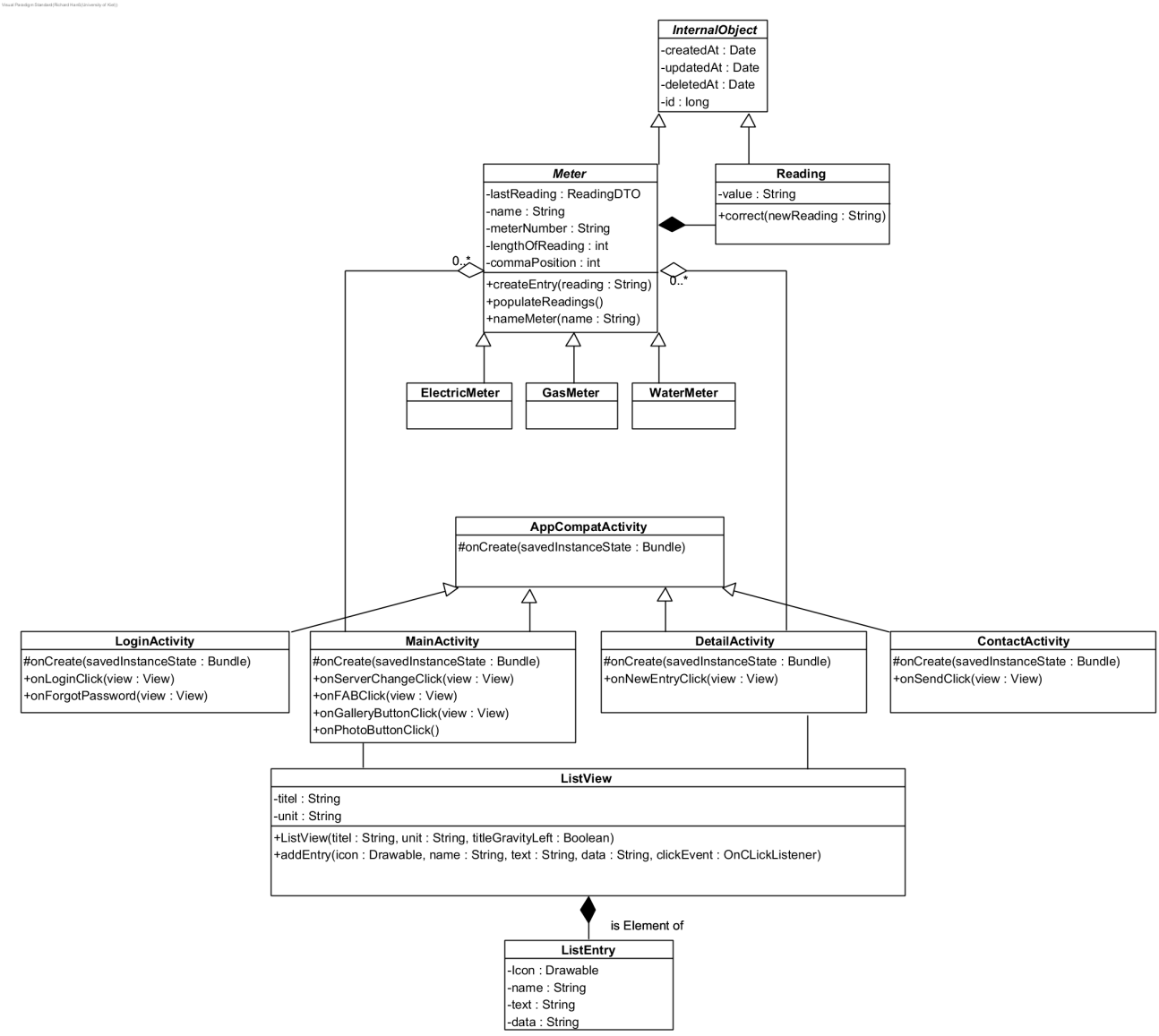


Abbildung 6.6: Klassendiagram - Android-View

Erklärung - View der App

Meter/InternalObject/Reading	Die hier aufgeführte Meter Klasse ist dieselbe, die auch in dem Logikdiagram dargestellt ist.
ElectricMeter/GasMeter/WaterMeter	Die hier aufgeführte Meter Klasse ist dieselbe, die auch in dem Logikdiagram dargestellt ist.
AppCompatActivity	Diese Klasse ist standardmäßig die Superklasse von allen Activities in Android Studio
LoginActivity	Diese Activity stellt UI-Elemente bereit, die es dem Nutzer erlauben sich anzumelden.
MainActivity	Auf dieser Activity kann der Nutzer seine Zähler und einige Zählerstände einsehen, sowie auf sie zugreifen.
DetailActivity	Auf dieser Activity kann der Nutzer Zählerstände hinzufügen und einsehen.
IssueActivity	Auf dieser Activity kann der Nutzer einen Admin kontaktieren, falls er ein Problem hat.
ListView	Diese Klasse erleichtert das hinzufügen und maintainen von Listen in der MainActivity und DetailActivity.
ListEntry	Diese Klasse stellt einen Listeneintrag dar.

Kapitel 7

Sequenzdiagramme

In diesem Kapitel sind die Sequenzdiagramme beschrieben, die Vorgänge beschreiben, deren Verhalten nicht trivial ist. Nicht trivial ist ein Verhalten, bei dem der Ablauf des Diagramms nicht einem "Durchreichen" von Aufrufen entspricht.

7.1 Triviale Abfrage

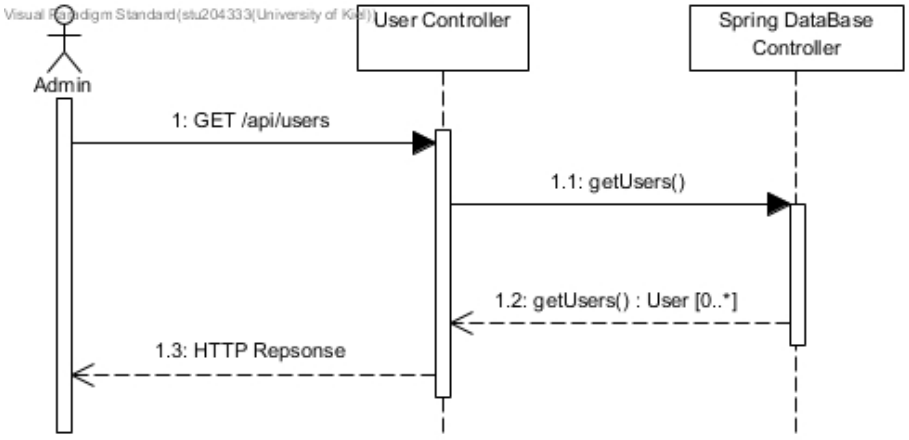


Abbildung 7.1: Abfrage der Nutzerlisten

Als Beispiel für ein triviales Sequenzdiagramm lässt sich das Abfragen der Daten durch einen Admin hernehmen. Dabei beschreibt "User [0..*]" die Rückgabe eines "UserList DTO".

7.2 Bilder in der App übermitteln

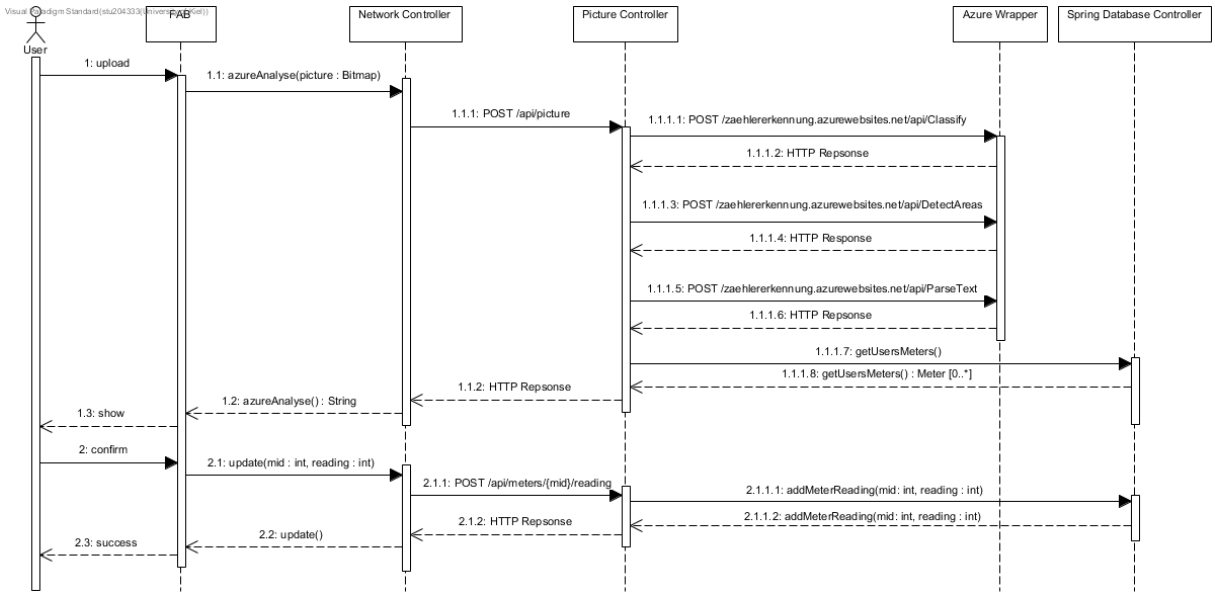


Abbildung 7.2: Erfolgreiche Bildübermittlung

Das folgende Sequenzdiagramm beschreibt den Vorgang des erfolgreichen Übermittelns eines Zählerstandes als Bild. Zuerst lädt der Nutzer ein Foto hoch, dieses wird mittels der Methode azureAnalyse() vom Netzwerk-Controller per POST-Request ans Back-End der Website geschickt. Dieses leitet das Bild per HTTP an die Klassifizierungsschnittstelle des AzureWrappers weiter. Nach der Antwort wird über HTTP das Bild an die nächste Schnittstelle, “Area Detect” geschickt. Als Letztes wird das Bild via HTTP an die Schnittstelle “Parse Text” geschickt. Die erhaltenen Daten werden jetzt einer ersten Plausibilitätsprüfung unterzogen. Die wird die Richtigkeit der Länge und der Klassifizierung überprüft.
Nun werden mittels getUsersMeters() alle Zähler für den Nutzer abgefragt, um zu Prüfen, ob die Nummer des Zählers der Nummer eines der Zähler des Users entspricht. Wenn dies der Fall ist, wird eine HTTP-Response an den Network-Controller gesendet, die die gelesene Zählernummer und den gelesenen Zählerstand übermittelt. Nun wird der Nutzer aufgefordert die gelesenen Werte zu bestätigen.
Der Ablauf eines manuellen Eintrages entspricht dem unteren Teil des Diagramms (ab Punkt 2).

7.3 Zählerstände für User updaten

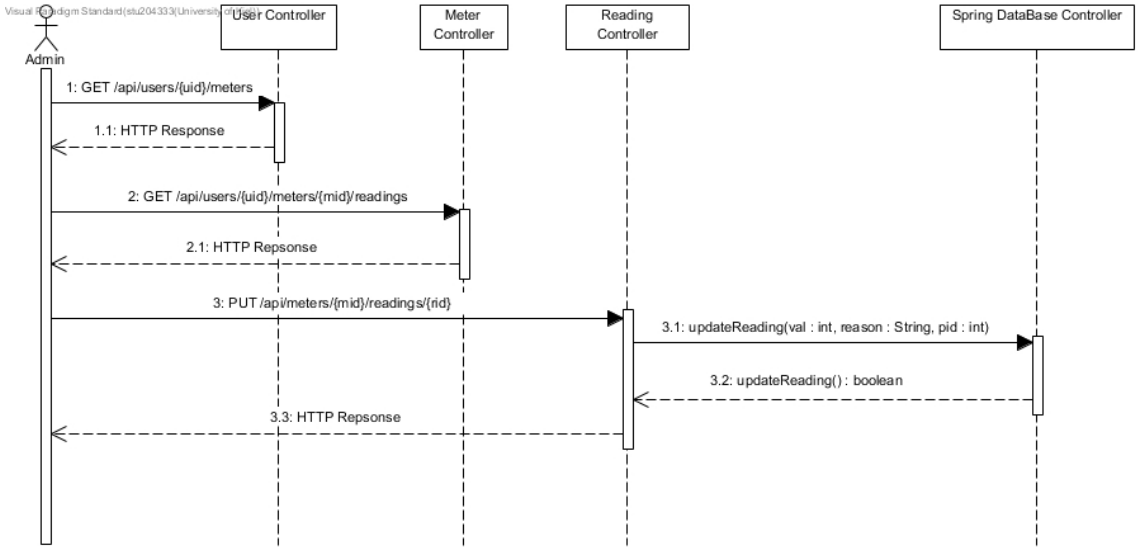


Abbildung 7.3: Zählerstand eines Nutzers als Admin updaten

Administratoren können unter Angabe eines Grundes Zählerstände aktualisieren. Dafür muss der Administrator zuerst mittels eines HTTP-Requests alle Zähler zu einer User-ID vom UserController abfragen. Danach werden mittels einer weiteren Anfrage alle Zählerstände des Zählers abgefragt. Nun kann mittels eines PUT-Request mit der Reading-ID ein bestimmter Eintrag verändert werden. Dafür wird der neue Zählerstand, der Grund der Änderung und die ID des Admins versendet, die dann für den Funktionsaufruf "updateReading" verwendet wird. Wenn die Änderung erfolgreich war, sendet der Reading Controller in der HTTP Response einen Erfolg zurück.

Kapitel 8

Glossar

Abkürzung	Beschreibung
Zähler	Bei einem Zähler handelt es sich um einen Gas-, Strom- oder Wasserzähler. Er misst den Verbrauch der jeweils namensgebenden Ressource.
FAB (Floating Action Button)	Ein FAB ist ein Knopf, der in der unteren rechten Ecke einer Android-App sitzt und Zugriff zu essentiellen Funktionen ermöglicht.
Administrator (Admin)	Der Administrator, vornehmlich ein Mitarbeiter von adesso energy, hat übergeordnete Zugriffsrechte, die es ihm erlauben, auf alle Daten zuzugreifen, sowie diese zu ändern. Er interagiert hierfür ausschließlich mit der Website.
Benutzer (User)	Als Benutzer werden Kunden von adesso energy bezeichnet. Sie haben die Möglichkeit, über die App oder Website auf ihre Nutzerdaten zuzugreifen und aktuelle Zählerstände hochzuladen. Dabei bietet die App die Option, einen Zählerstand automatisch aus einem Bild zu erkennen. Der Benutzer kann sowohl mit der App als auch mit der Website interagieren.
Web-Applikation	Der Server ist nicht Teil einer Web-Applikation. Aufgrund der umfassenden React-Architektur betrachten wir die Website als eigene Applikation.
DTO (data transfer object)	Als data transfer objects bezeichnen wir die Objekte, die unsere Anwendungen über das HTTP-Protokoll übertragen.
Paging	Durch paging wird beschränkt, wie viele Daten ein Server auf einmal an einen Client versendet.

Tabelle 8.1: Glossar