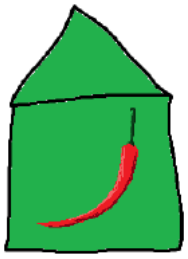
Duale Hochschule Baden-Württemberg Mannheim

Automated Greenhousearbeit

TINF22IT1

Verfasser(in): Jan Hampel
Matrikelnummer: 1234



1 Introduction

Als duale Studierende der DHBW-Mannheim haben wir eine große Leidenschaft für Software Engineering in den Bereichen Frontend, Backend und Embedded. Hinzu kommt ein großes Interesse für Botanik, insbesondere für den Anbau von Chili. Während die erste Leidenschaft auch in den Praxisphasen in Köln, Augsburg, Braunschweig oder Hamburg problemlos ausgelebt werden kann, sieht es bei der zweiten Leidenschaft schon etwas schwieriger aus. Da wir jeweils nur drei Monate an einem Standort sind, ist es uns leider nicht möglich, uns kontinuierlich um die Pflege und das Wachstum unserer Chilipflanzen zu kümmern.
Die Vorlesung Software Engineering hat uns jedoch dazu inspiriert, diese beiden Leidenschaften zu vereinen, um das Problem der Abwesenheit und der daraus resultierenden mangelnden Pflege der Pflanzen zu lösen.
Die Idee ist, mit Hilfe von Soft- und Hardware ein "Automated Greenhouse" zu entwickeln, das unsere Pflanzen während der Praxisphasen pflegt. Um einen möglichst hohen Automatisierungsgrad in der Pflanzenpflege zu erreichen, müssen zunächst die Anforderungen an das Projekt definiert werden. Da die Leidenschaft für Software Engineering auch in das Projekt einfließen soll, haben wir uns dafür entschieden, die Anforderungsdefinitionen mit Hilfe von User Stories umzusetzen.

1.1 User Story 1: Automatische Bewässerung

Da wir alle zusammen in einer WG wohnen, haben wir das Problem, dass während der Praxisphasen niemand in der WG ist, der eventuell kleinere pflegerische Tätigkeiten, wie z.B. das Gießen, übernehmen könnte. Daraus ergibt sich die erste User Story:
Als Gewächshausbesitzer
möchte ich dass die Bewässerung meines Gewächshauses automatisch erfolgt,
damit die Pflanzen auch ohne meine Anwesenheit und meine Zeitinvestition bestmöglich bewässert werden

Akzeptanzkriterien:

- Es wird die Feuchtigkeit des Bodens gemessen und an den Gewächshauscontroller übertragen.
- Fällt die Feuchtigkeit unter einen eingestellten Schwellwert, so wird eine Pumpe oder Ventil aktiviert und der Boden befeuchtet.

- Die Bewässerungsdauer und -intensität sollen anhand der Bodenfeuchtigkeitsdaten angepasst werden. Bei niedrigerer Feuchtigkeit sollte mehr bewässert werden.
- Es wird nicht überwässert. Sobald der Boden befeuchtet wurde, wird eine gewisse Zeit vor der nächsten Messung abgewartet.

Aus dieser User Story geht hervor, dass wir einen Controller benötigen, der über Sensoren und Aktoren mit der Pflanze interagieren kann. Der Controller muss in der Lage sein, einen Feuchtigkeitssensor abzufragen, eine Pumpe anzusteuern und den Feuchtigkeitswert auf einen einstellbaren Wert zu regeln.

1.2 User Story 2: Licht Steuerung

Da wir die Pflanzen in unseren Zimmern lassen wollen und die Rollläden während unserer Abwesenheit schließen, müssen wir die Pflanzen mit künstlichem Licht versorgen.

Als Gewächshausbesitzer der gerne die Rolläden geschlossen hat,
möchte ich dass meine Pflanzen mit künstlichem licht versorgt werden,
damit die Pflanzen auch ohne meine Anwesenheit einen optimalen Lichtzyklus haben und gesund wachsen können.

Akzeptanzkriterien:

- Eine Lampe soll zu bestimmten Zeiten ein- und ausgeschaltet werden.
- Die Zeiten, zu denen die Lampe ein- und ausgeschaltet wird, sollten auch geändert werden können, ohne dass der Gewächshausbesitzer physisch ist.

Aus dieser User Story geht hervor, dass der Controller in der Lage sein muss, eine Lampe ein- und auszuschalten. Die Tatsache, dass die Zeitwerte auch aus der Ferne geändert werden können sollen, zeigt, dass der Gewächshausbesitzer noch mit dem Controller interagieren können muss.

1.3 User Story 3: Anzeige von Echtzeit-Messdaten

Als Gewächshausbesitzer
möchte ich die Umgebungsparameter in meinem Gewächshaus aus der Ferne überwachen können,

damit ich frühzeitig erkenne, wenn die Wachstumsbedingungen nicht optimal sind, selbst wenn ich physisch nicht anwesend bin.

Akzeptanzkriterien:

- Die Messdaten, die der Controller erfasst, wie die Temperatur und die Bodenfeuchtigkeit sollen für den Gewächshausbesitzer einsehbar sein.
- Die Daten sollen von einem mobilen Endgerät aus abrufbar sein.
- Die Daten sollen dabei dem Gewächshausbesitzer einfach und verständlich präsentiert werden.

1.4 User Story 4: Remote Controll

Als Gewächshausbesitzer
möchte ich dazu in der Lage sein alle Parameter der Pflege, die beeinflussbar sind, auch aus der Ferne verändern zu können,
damit ich die Wachstumsbedingungen für meine Pflanzen optimieren und ihr Wohlbefinden sicherzustellen kann, selbst wenn ich physisch nicht anwesend bin.

Akzeptanzkriterien:

- Die Zeiten, zu der die Lampe eingeschaltet ist, soll aus der Ferne einstellbar sein.
- Die Feuchtigkeit, die die Erde meiner Pflanze haben soll, soll aus der Ferne einstellbar sein.
- Die Parameter sollen durh den Gewächshausbesitzer einfach und verständlich einstellbar sein.

1.5 Team Bildung

Aus den User Stories lässt sich ableiten, dass ein Controller benötigt wird, der die Sensoren abfragt und die Aktoren ansteuert. Da die Sensorwerte für den Gewächshausbesitzer auch aus der Ferne einfach und verständlich über ein mobiles Endgerät abrufbar sein sollen, haben wir uns dafür entschieden, die Werte auf einer Webseite zu visualisieren. Dazu muss der Controller in der Lage sein, diese Werte an einen Server zu senden, von dem die Website die Werte abrufen kann. Hierfür wird ein Backend benötigt, das auch dazu verwendet werden kann, die Parameter Zeit und Feuchte, die auf der Website

eingestellt werden können, an den Controller zu übergeben.
Wir haben beschlossen, unsere Gruppe in drei Teams aufzuteilen. Ein Team kümmert sich um die Entwicklung der Hardware, ein Team um die Entwicklung des Frontends und das letzte Team um die Entwicklung des Backends.

2 Methodik

Dieses Kapitel fokussiert sich auf die Zusammenarbeit innerhalb der Gruppe und die verwendeten Methoden und Verfahren. Insbesondere wird auf die inkrementelle Softwareentwicklung beleuchte, welche als wesentliches Modell innerhalb unseres Projektes angewendet wurde. Es wurde aufgrund der Einteilung in Teams gewählt, welche maximal aus 2 Mitgliedern bestanden. Ein agiles Modell wäre im diesem Zusammenhang nicht sinnvoll.

inkrementelle Softwareentwicklung

Zunächst gab es eine Initiale Planungsphase, in der die wichigsten Ziele und Pläne festgehalten wurden. Diese können. Unsere Aufgaben und Features kann man in meherere kleinere Entwicklungszyklen unterteilen. Dabei besteht ein Entwicklungszyklus aus der Entwicklung und Ausarbeitung von Anforderungen und der Analyse und Design, um neue Features in die bestehende Software in die bestehende Umgebung einbinden zu können. Anschließend werden die Features implementiert und getestet. Die einzelnen Teile des Zyklus werden im Folgenden näher beleuchtet.

Initiale Projektplanung

Die Projektidee und die übergeordneten Ziele wurden bereits in der Einleitung beleuchtet.

Anforderungsphase

Am Anfang des Entwicklungszykluses werden zunächst die Anforderungen erfasst. Dabei orientierte sich das Ziel der Anforderungen an den Fortschritt im Projekt. Die ersten Anforderungen galten dabei zunächst dem Aufbau des Grundgerüsts der Hard- und Software, um eine anschließende Entwicklung möglichst simpel zu halten. In diesem Zuge wurde beispielsweise die grundlegende Seitenstruktur des Frontends erstellt oder die Datenbank im Backend aufgesetzt. In späteren Entwicklungszyklen wurden dann Anforderungen an das konkrete Design im Frontend oder die konkreten API-Funktionalitäten erarbeitet.
Die sehr diversen Anforderungen an ein Feature wurden gesammelt. Meist diente initial eine Mindmap als Grundlage für die spätere Konkretisierung der Anforderungen. Da die unterschiedlichen Teams zumeist Features unabhängig voneinander entwickelt haben, wurden unterschiedliche Methoden verwendet, um die Anforderungen zu dokumentieren. Eine Methode waren

Issues, bei denen die einzelnen Gruppenmitglieder zu bestimmten größeren und kleineren Aufgaben zugewiesen wurden.

Designphase

In der Designphase eines Entwicklungszyklus wurde ein Plan zum Einbinden der Features in die bestehende Umgebung erarbeitet. Neben der primären Aufgabe die Anforderungen umzusetzen, geht es bei dem Design auch um die spätere Erweiterbarkeit der Software. Da es sich bei den Anforderungen um ein sehr heterogenes Feld handelt, kann man keine konkrete Methodik angeben, wie die Umsetzung der Anforderungen geplant wurde. Im Folgenden werden bespielhaft einige Methoden auf jedem Team vorgestellt:

- **Hardware**
  Um in der Hardware die Anforderungen umzusetzen, wurden Skizzen per Hand erstellt, um die Systemarchitektur modellieren und ändern zu können. Eine beispielhafte Skizze kann man im Folgenden sehen.

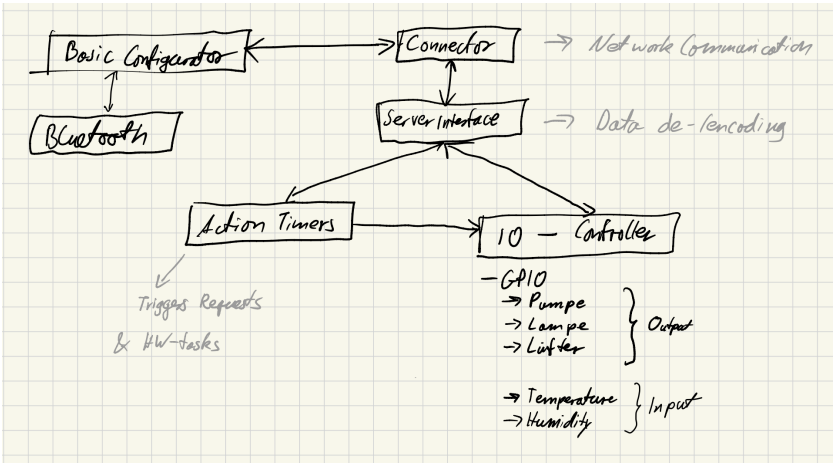


Figure 1: Beispielhafte Skizze zur Umsetzung von Hardware-Anforderungen

  Anhand dieser Zeichnung gelang es komplexe Sachverhalte auf das Wesentliche zu reduzieren und so die Implementierung zu planen.

- **Frontend**
  Im Frontend wurden beispielsweise Mockups erstellt, um die Änderung am User Interface zu planen. So konnten beispielsweise Proportion von Graphelementen zu Verwaltungsleiste auf der Datenseite der Webanwendung modelliert werden. Dadurch wurde die Implementierung erleichtert, da man die Anordnung der Oberflächenelemente nicht in

der Entwicklung anpassen muss. Die dafür verwendete Software war Keynote.

- **Backend** Das Backend beinhaltet vergleichweise wenig Programmlogik und regelt hauptsächlich den Datenbankzugriff. Deshalb wurde sich für eine Schichtenarchitektur entschieden. Hier werden zwei Schichten, eine Datenbankschicht und eine Serviceschicht verwendet. Die Datenbankschicht stellt wiederverwendbare Funktionen für den Datenbankzugriff bereit. Verwendet werden diese Funktionen dann von den Rest-Endpunkten, welche in sogenannte Services der Serviceschicht aufgeteilt sind.

  Diese Schichtenarchitektur ist anschaulich in Abbildung 2 dargestellt. Sie bietet so einige Vorteile, wie die leicht Erweiterbarkeit und Maintainability durch gute Übersichtlichkeit.

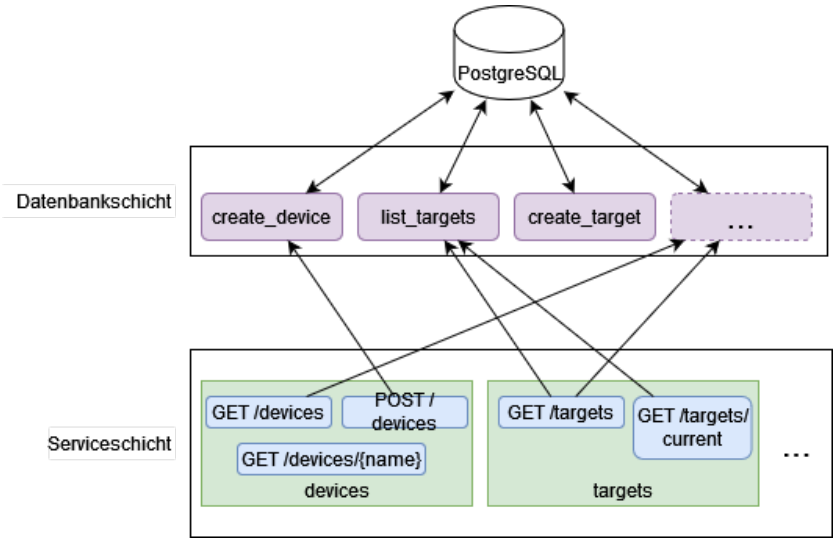


Figure 2: Schichtenarchitektur des Backends. Hervorgehoben sind REST-Endpunkte (blau), Services (grün) und Datenbank-Funktionen (lila).

Implementierung

Den erarbeiteten Plan gilt es in dieser Phase eines Zyklus umzusetzen. Die vorherigen Arbeitsabläufe halfen vor allem dabei, die Implementierung zu beschleunigen. Innerhalb dieser Phase wurden ebenfalls in den einzelnen Teams unterschiedliche Verfahren verwendet. Besonders hervorzuheben waren Kanban-Boards, die sowohl im Hardware- und Frontend-Team verwendet.

So konnte ein schneller Überblick über die im Team bearbeiteten Themen gewonnen werden. Gerade bei der Implementierung von einzelnen Features half das besonders größere Probleme in kleine handhabbare aufzuteilen.
Da das Backend ausschließlich von Florian erstellt wurde, hat er sich dafür entschieden die anstehenden Aufgaben mittels einer einfachen txt-Datei zu dokumentieren. Das hatte den Vorteil, dass die interne Entwicklung an diesem Teil im Projekt schlank gehalten wurde.

Testung

Die Testung der implementierten Features im Projekt gilt es im Anschluss zu testen. Die Testung ist ein wesentlicher Bestandteil, der in dem entsprechende Abschnitt näher beleuchtet wird. Aus diesem Grund wird hier nicht mehr näher darauf eingegangen.

Evaluation

In wöchentlichen Meetings während der Software Engineering Vorlesung wurde sich getroffen und frei über das Projekt gesprochen. In diesem Zusammenhang ging es vor allem um den Fortschritt, der in der letzten Woche erzielt wurde und was für die nächste Woche geplant ist. In diesem Zusammenhang konnten die Ergebnisse der einzelnen Entwicklungszyklen hinterfragt werden. Dieses Feedback galt dann entweder der Verbesserung aktueller Ergebnisse oder des nächsten Entwicklungszyklus.
Darüber hinaus galten diese Meetings auch der Kommunikation zwischen den Teams. Bei unserem Automated Greenhouse ist ein wesentlicher Bestandteil die Zusammenarbeit zwischen den Teams im Projekt. Die unterschiedlichen Aufgabenbereiche müssen so aufeinander abgestimmt werden, dass die Teams trotz Differenzen im Fortschritt weiter an ihren Aufgaben arbeiten können. Beispielhaft kann man hier die Endpunkte der API nennen. Diese verbinden das Frontend mit den Sensoren und Aktoren der Hardware und werden von dem Backend implementiert und bereitgestellt. Man erkennt, dass die Entwicklung sehr stark voneinander abhängt und deshalb die Meetings sich zu einem sehr wichtigen Bestandteil unseres Vorgehen entwickelt haben.

3 Prozessdiagramm des Backends

In diesem Kapitel werden die REST-Endpunkte des Backends und ihre Funktionsweise mithilfe von Prozessdiagrammen aufgezeigt. Da das Backend in Rust, einer nicht-objektorientierten Sprache, implementiert wird, ist ein Klassendiagramm nicht sinnvoll.

Die Funktion des Backends besteht darin, verschiedene REST-Endpunkte bereitzustellen. Es werden hier acht Endpunkte definiert, die in drei Services (bzw. Gruppen) unterteilt werden können: devices, datapoints und targets. Diese Aufteilung ermöglicht eine bessere Code-Wiederverwendung innerhalb eines Services und vereinfacht außerdem die Prozessdiagramme.

Vor Beginn der Implementierung wurden die Endpunkte in Textform formuliert. Zur besseren Anschaulichkeit werden hier die fertigen Endpunkte betrachtet, die in Abbildung 3 basierend auf der generierten OpenAPI-Dokumentaion mithilfe von SwaggerUI dargestellt sind [1]. Im Folgenden wird für jeden Service ein Prozessdiagramm vorgestellt.

[1] Vollständige Dokumentation inkl. Datentypen ist unter https://se-backend.fly.dev/swagger-ui einsehbar

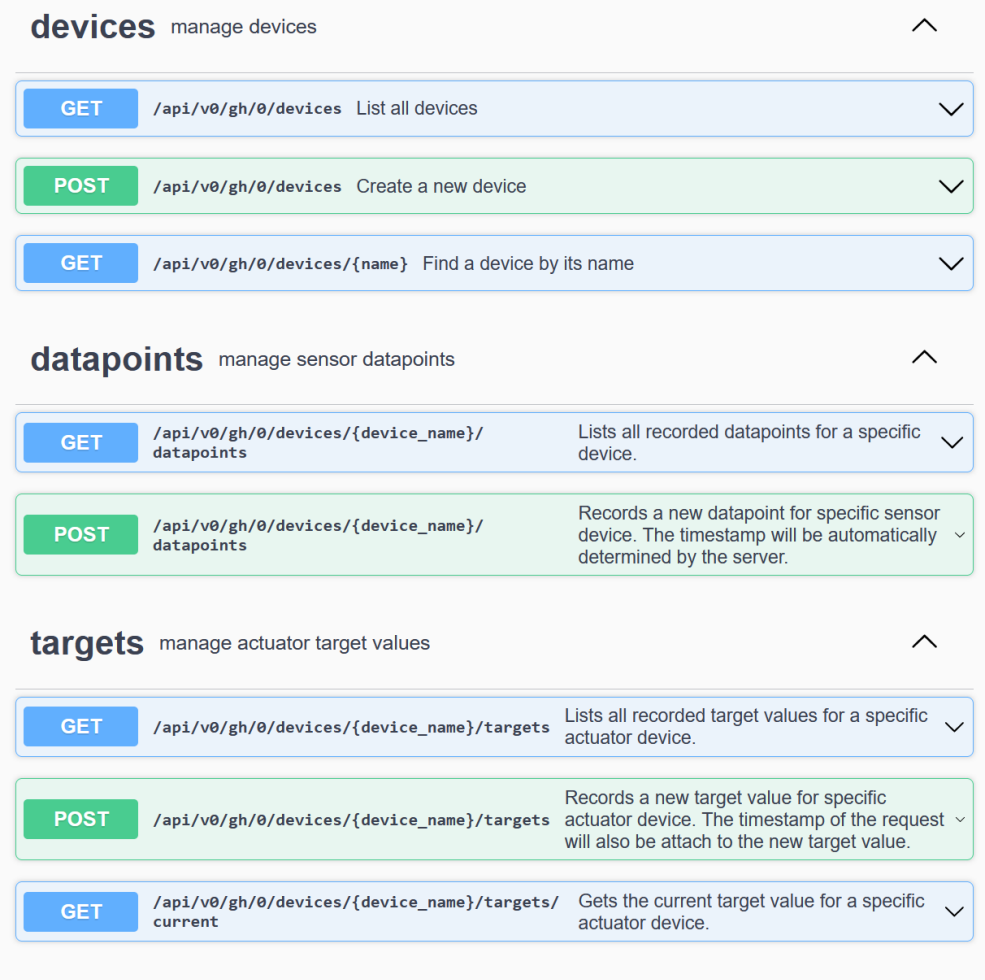


Figure 3: REST-Endpunkte (Rendering der OpenAPI-Dokumentaion durch SwaggerUI)

3.1 devices-Service

Der devices-Service umfasst mehrere Endpunkte zum Erstellen oder Abfragen von Devices, also Sensoren oder Aktoren der Hardware. In Abbildung 4 sind die nötigen Prozesse dargestellt.

Zuerst wird bei den Endpunkten (in blau), welche JSON-Parameter akzeptieren, überprüft, ob die übergebenen JSON-Felder valide sind. Falls dies nicht der Fall ist, wird ein entsprechender Fehlerstatuscode (hier: 400 Bad Request) zurückgegeben. Diese Überprüfung wird in den folgenden beiden Unterkapiteln zu den anderen Services weggelassen, ist aber dennoch überall nötig, wo es JSON-Parameter gibt.

Im Anschluss wird bei allen drei Endpunkten eine Anfrage an die Datenbank gemacht und dann entweder das Ergebnis oder ein Fehlerstatuscode zurückgegeben.

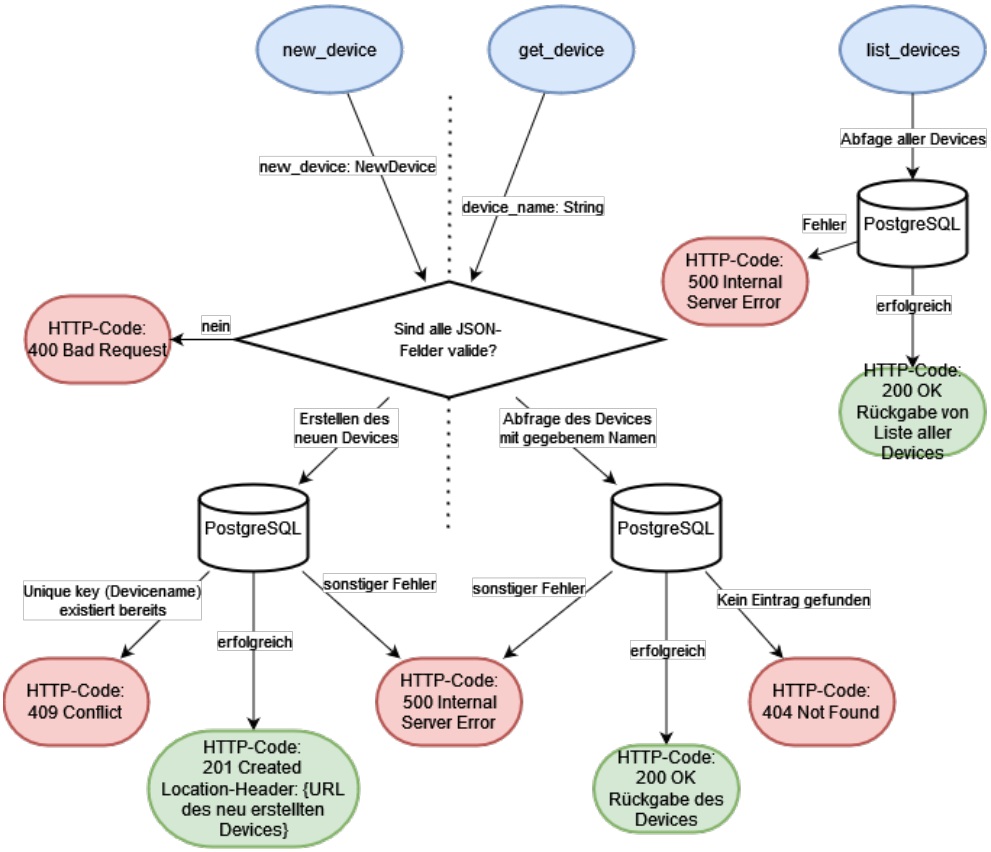


Figure 4: Prozessdiagram der Endpunkte im devices-Service

3.2 datapoints-Service

Der datapoints-Service verwaltet die Datenpunkte eines Sensor-Devices. Hier sind zwei Endpunkte zum Erstellen eines neuen Sensor-Datenpunktes und zum Abfragen aller Datenpunkte nötig. Beide haben gemeinsam, dass das angefrage Device vom Typ Sensor sein und existieren muss, weshalb im Prozessdiagramm in Abbildung 5 die oberen Prozesse zusammengefasst sind. Nach der Device-Typ Überprüfung wird in beiden Fällen eine weitere Anfrage an die Datenbank gemacht und das Ergebnis oder ein Fehlerstatuscode zurückgegeben.

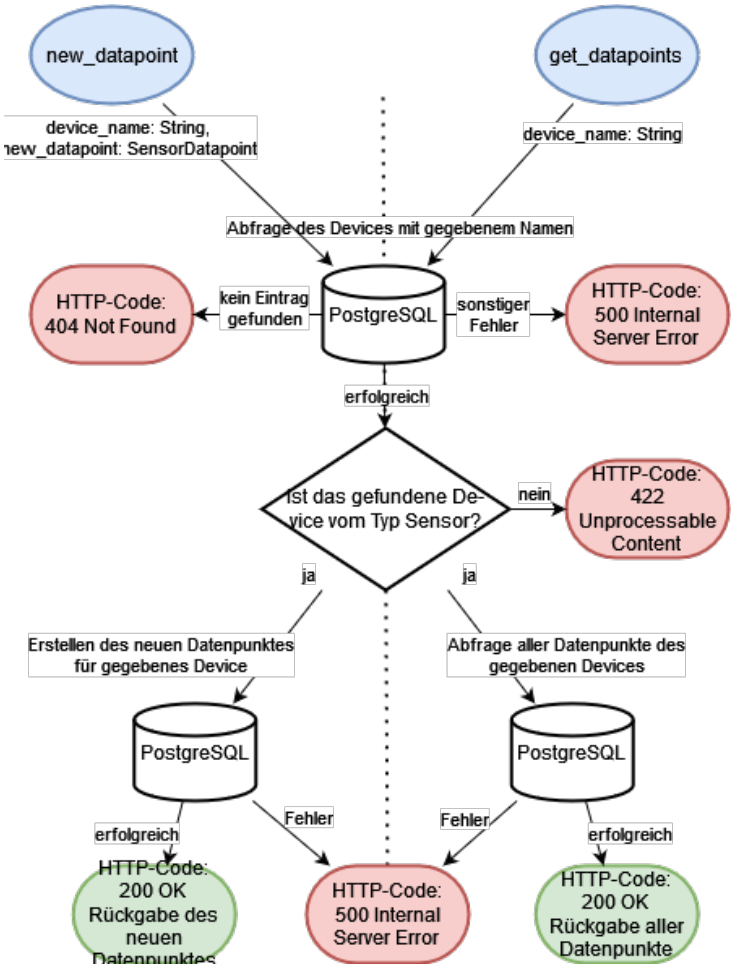


Figure 5: Prozessdiagram der Endpunkte im datapoints-Service

3.3 targets-Service

Der targets-Service ist für die Endpunkte zuständig, welche die Zielwerte (TargetValues) von Actuator-Devices setzen und abfragen. Es werden wie bei den Sensor-Datenpunkten zwei Endpunkte zum Erstellen und Abfragen aller Targets und zusätzlich ein Endpunkt zum Abfragen des aktuellsten Zielwertes angeboten [2].

Auch hier erfolgt zuerst eine gemeinsame Überprüfung, ob das angefrage Device existiert und ein Aktuator ist. Daraufhin wird je nach Endpunkt

[2]So kann die Hardware, welche nur begrenzt Ressourcen besitzt, den aktuellsten Wert abfragen. Außerdem ist denkbar, dass das Backend in der Zukunft automatisch Zeitpläne auflöst, ohne dass die Hardware angepasst werden muss.

eine Datenbank-Anfrage getätigt und das Ergebnis oder ein Fehlerstatuscode zurückgegeben.

Figure 6: Prozessdiagram der Endpunkte im targets-Service

4 Einblick in den Backend-Quellcode

Dieses Kapitel gibt einen Einblick in den Quellcode des Backends. Dazu wird ein bestimmer REST-Endpunkt betrachtet:

POST /api/v0/gh/0/devices/[device_name]/datapoints

Er ist dafür verantwortlich, einen neuen Sensor-Datenpunkt aufzunehmen, wird also regelmäßig von der Hardware aufgerufen. Der URL-Pfad ist zusammengesetzt aus vier Teilen:

- /api/v0: Versionierung der API.
- /gh/0: Es wird das Greenhouse (gh) mit Index 0 angesprochen[3].
- /devices/[device_name]: Es wird das Sensor-Device mit gegebenem Namen betrachtet.
- /datapoints: In Verbindung mit dem HTTP Operator POST heißt dies, dass ein neuer Datenpunkt erstellt werden soll.

Hier hat allerdings nur der Teilpfad /devices/[device_name]/datapoints Relevanz. Er beinhaltet ein Argument des REST-Endpunktes: Den Namen des Sensor-Devices.

In Kapitel 2 wurde bereits die Schichtenarchitektur des Backends präsentiert. Diese spiegelt sich im Quellcode gut sichtbar wieder. Zuerst wird die nötige Funktion in der Datenbankschicht betrachtet. Ihr Rust-Code ist in Abbildung 7 zu sehen. In Zeile 1-5 befindet sich der Funktionskopf, welche als Parameter die Datenbankverbindung (conn), den neuen Sensor-Datenpunkt und den Sensor-Devicenamen definiert. Nach einer ersten Typ-Umwandlung folgt der Bau des INSERT SQL-Queries mithilfe der Rust-Bibliothek diesel [4]. Diesel ist ein ORM (object-relational mapping) tool, um u.a. SQL-Queries direkt aus Rust-Code zu bauen. Einige Vorteile sind die Typ-Sicherheit oder ein automatischer Schutz vor SQL-Injections. Zuletzt wird dann, falls der Datenbankaufruf erfolgreich war, der neue Sensor-Datenpunkt zurückgegeben.

[3] Dies ist ein Platzhalter für den zukünfigten Multi-Greenhouse Betrieb.
[4] siehe https://diesel.rs/

```rust
pub fn create_datapoint(
    conn: &mut PgConnection,
    datapoint: SensorDatapoint,
    sensor_name: String,
) -> Result<SensorDatapoint, DatabaseError> {
    // Umwandlung zu Datenbank-struct
    let db_datapoint = DbSensorDatapoint::from_model(
        datapoint,
        sensor_name
    );

    // Bau und Ausführung eines SQL-Queries mit diesel
    // Der ?-Operator bricht die Funktion bei einem Fehler ab.
    let db_datapoint = db_datapoint
        .insert_into(super::schema::sensor_datapoints::table)
        .returning(DbSensorDatapoint::as_returning())
        .get_result(conn)?;

    // Rückgabe des neuen Sensor-Datenpunkes
    Ok(db_datapoint.into())
}
```

Figure 7: Rust-Funktion der Datenbankschicht zur Erstellung eines neuen Sensor-Datenpunktes

Als nächstes wird die Funktion der Serviceschicht vorgestellt, welche den zuvor genannten REST-Endpunkt bereitstellt. Ihr Quellcode ist in Abbildung 8 dargestellt. Der Funktionskopf von Zeile 1 - 5 ist hier von besonderer Bedeutung. Die Rust-Bibliothek axum[5] bietet eine Art Framework für die Erstellung von HTTP-Servern. Sie ist auch dafür verantwortlich, dass die Funktionsparameter richtig gesetzt werden. Es werden von ihr also die Datenbankverbindung (Zeile 2), der Sensor-Name aus dem URL-Pfad (Zeile 3) und die übermittelten JSON-Daten (Zeile 4) überliefert.

Die Funktion stellt dann die Datenbankverbindung her (Zeile 7), überprüft mit einer Hilfsfunktion ob es sich bei dem angegebenen um ein valides Sensor-Device handelt (Zeile 10) und verpackt erstellt ein neues SensorDatapoint-struct (Zeile 13-16). Dann wird die zuvor vorgestellte Funktion der Datenbankschicht aufgerufen (Zeile 19-23) und je nach Rückgabewert der neu erstelle Datenpunkt oder ein Fehler zurückgegeben.

[5] siehe https://github.com/tokio-rs/axum

```rust
async fn new_datapoint(
    State(AppState { database_pool, .. }): State<AppState>,
    Path(sensor_name): Path<String>,
    Json(new_datapoint): Json<NewSensorDatapoint>,
) -> ApiResult<SensorDatapoint, String> {
    // Herstellen einer Datenbank Verbindung
    let mut database_connection = database_pool.get().unwrap();

    // Überprüfung: Existiert das Device und ist es vom Typ Sensor?
    check_device_sensor_type(&mut database_connection, &sensor_name)?;

    // Erstellen des eigentlichen Datapoint-structs
    let sensor_datapoint = SensorDatapoint {
        timestamp: Utc::now(),
        value: new_datapoint.value
    };

    // Aufruf der zuvor beschriebenen Funktion der Datenbankschicht
    match database::sensor_datapoint::create_datapoint(
        &mut database_connection,
        sensor_datapoint,
        sensor_name,
    ) {
        // Falls Datenbank-Aufruf erfolgreich:
        Ok(datapoint) => Ok(
            ApiSuccess::new_with_status(
                StatusCode::OK,
                datapoint
            )),
        // Falls zu dem aktuellen Zeitpunkt schon ein Datenpunt existiert:
        Err(DatabaseError::UniqueKeyViolation) => Err(
            ApiError::new_with_status(
                StatusCode::CONFLICT,
                "Failed to create new datapoint.
                Wait a few seconds before trying again.".to_owned(),
            )),
        // Sonstiger Fehler:
        Err(err) => Err(ApiError::new_unknown(format!("{err:?}"))),
    }
}
```

Figure 8: Rust-Funktion der Serviceschicht, welche den Endpunkt zum Erstellen eines Datenpunktes bereitstellt.

Es ist außerdem wichtig zu erwähnen, dass durch erfolgte Abstraktion zum Teil unsichtbare Programmlogik stattfindet. Wird bspw. wie in Zeile 38 ein unbekannter ApiError erstellt, wird der Fehler geloggt und der HTTP-Statuscode 500 "Internal Server Error" zurückgegeben. Das gleiche passiert auch bei jeglichen Rust-Panics, welche sich ähnlich wie Exceptions verhalten. So lässt sich der Code ohne Wiederholungen auf die wirklich nötige Programmlogik reduzieren.

5 Use-Case

Der folgende Use-Case beschreibt eine Interaktion des Benutzers mit dem Frontend für das Anzeigen von Messdaten und das Einstellen von neuen Kontrollwerten, wie in Bild 9 dargestellt.

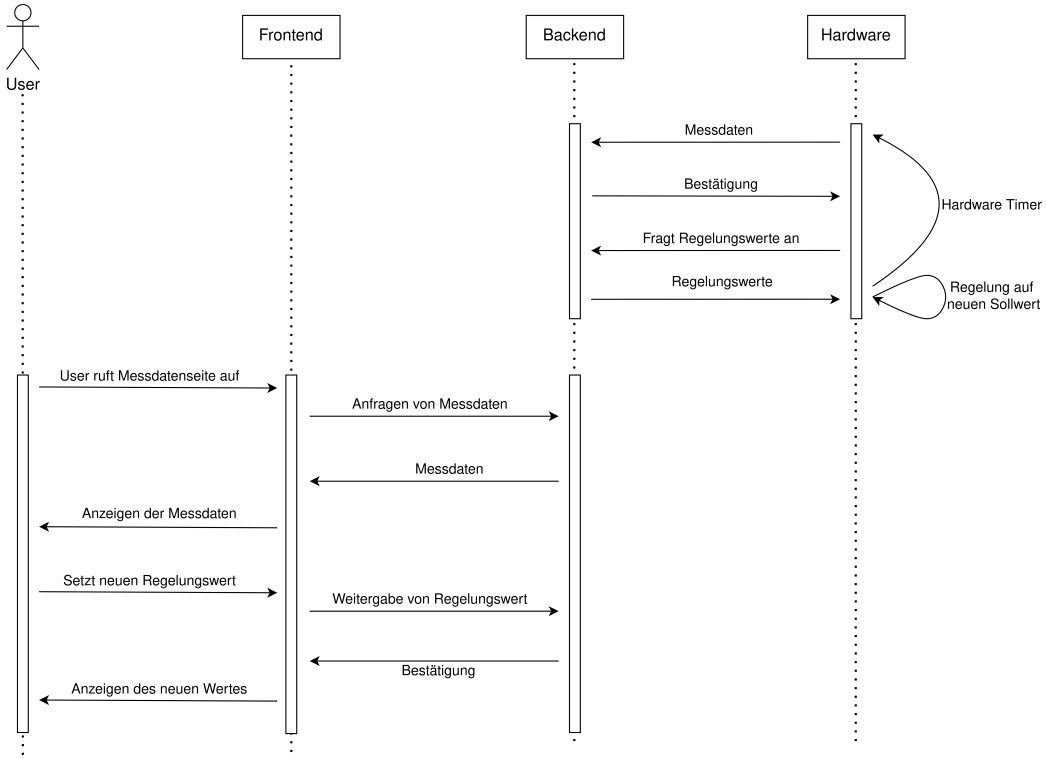


Figure 9: Sequenzdiagramm einer Interaktion

In Bild 9 kann besonders die Asynchronität der Interaktionen zwischen Frontend ↔ Backend und Backend ↔ Hardware gesehen werden. Das Backend ist der Dreh- und Angelpunkt des ganzen Systems, woraus sich diese Beziehungen ergeben: die Hardware und das Frontend kennen das Backend, das Backend selbst macht jedoch keine Annahmen über über die anderen Systeme. Von beiden Seiten wird also ein Pull-Ansatz verfolgt.
Die Hardware kommuniziert mit dem Backend in periodischen Abständen. Hierbei sendet die Hardware die neue Sensordaten an das Backend (1) und fragt die neusten Kontrollwerte für Aktoren ab (2). Falls neue Kontrollwerte vorhanden sind, startet die Regelung der Aktoren zu diesem Kontrollwert hin.
Die eben gesendeten Sensordaten werden vom Backend in einer Datenbank

gespeichert, um sie dem Benutzer auf Abfrage anzuzeigen (3). Sobald der Benutzer auf der Weboberfläche einen neuen Kontrollwert für einen Aktor einstellt, wird dieser vom Frontend an das Backend gesendet (4), um dort gespeichert, und auf Abruf an die Hardware weitergegeben zu werden

6 Testplanung

Außerhalb von Modultests, welche in allen drei Komponenten durchgeführt wurden, wurden Systemtests durchgeführt. Auf diese wird sich in diesem Abschnitt fokussiert. Da alle Interaktionen der Hardware oder des Frontends über die API gehen müssen, steht diese bei der Planung und Betrachtung von Tests im Mittelpuntkt. Zuerst wird ein Überblick gegeben, wie die Tests geschrieben werden und im Anschluss wird etwas näher auf die Testplanung eingegangen.

Die Tests werden im Stil von Akzeptanztests aus Behavior Driven Development in der Sprache Gherkin verfasst. Dies hat den Grund, dass sich die Tests so einfacher formulieren und verstehen lassen. Da sich diese Tests auch ausführen lassen, sofern alle Schritte implementiert sind, geht man hier in die Richtung einer ausführbaren Spezifikation.

Um das Verhalten der API leichter zu testen, wurde eine virtuelle Python Umgebung geschrieben, welche ein Greenhouse simuliert. Diese Umgebung kann direkt aus den Tests genutzt werden, um der API Daten zu schicken und die Antworten dieser zu verifizieren.

Im folgenden wird die Planung der Tests und damit einhergehende Bildung der Äquivalenzklassen beispielhaft an API Tests vorgestellt.

6.1 Registrieren von Geräten

Wenn ein neuer Sensor oder ein neuer Aktor zu einem Greenhouse hinzugefügt wird, soll dies auch der API mitgeteilt werden. Hierfür gibt es einen Endpunkt, mit dem man ein Gerät unter einem bestimmten Namen registrieren kann. Dabei ergeben sich nur zwei Äquivalenzklassen: „ein Gerät mit diesem Namen existiert bereits" (GE im folgenden) und „ein Gerät mit diesem Name existiert noch nicht" (GNE im Folgenden):

Äquivalenzklasse	GE	GNE
Gültigkeit	Nicht Gültig	Gültig
Erwarteter Wert	HTTP 409	HTTP 201

Eine Implementierung des Verhaltens der Klasse GE in der Sprache Gherkin könnte folgendermaßen aussehen:

```
Scenario: Creating a device with a name that already exists
    Given a device with name soil_moisture exists
    When I try to create a device with the name soil_moisture
    Then the API should emit a negative response with HTTP code 409
```

Figure 10: Beispielhafte Testimplementierung in Gherkin

6.2 Senden von Datenpunkten

Da sich im letzten Beispiel nur zwei Äquivalenzklassen ergaben, welche sich nicht kombinieren lassen, wird im Folgenden ein etwas komplizierteres Beispiel betrachtet. Beim Senden von Datenpunkten muss wieder der Name eines Geräts angegeben werden, jedoch muss überprüft werden ob das Gerät vom Typ Sensor ist. Dies hat den Grund, dass Daten nur von Sensoren gesendet werden dürfen.

Hierbei ergeben sich folgende Äquivalenzklassen: „ein Gerät mit dem Namen existiert" (GE), „ein Gerät mit dem Namen existiert nicht" (GNE), „das Gerät ist vom Typ Sensor" (TS) „und das Gerät ist nicht vom Typ Sensor" (NTS). Diese Äquivalenzklassen lassen sich nun auf folgende Weisen kombinieren:

	TS	NTS
GE	GE+TS	GE+NTS
GNE	GNE	

Die unteren Beiden kombinierten Einträge können zu einem Eintrag zusammengefasst werden, da die Überprüfung nach Existenz wichtiger ist als nach richtigem Typ. Falls kein Gerät mit diesem Namen existiert, ist die Information, ob es vom richtigen Typ ist, überflüssig.

Für die zusammengesetzten Äquivalenzklassen kann man nun die Gültigkeit und die erwarteten Werte erfassen:

Äquivalenzklasse	GE+TS	GE+NTS	GNE
Gültigkeit	Gültig	Nicht Gültig	Nicht Gültig
Erwarteter Wert	HTTP 200	HTTP 422	HTTP 409

Eine mögliche Implementierung eines Tests zur Äquivalenzklasse GE+NTS könnte folgendermaßen aussehen:

```
Scenario: A device of type actuator tries to record a datapoint
    Given a device with name lamp exists
    And the device is of type actuator
    When the device tries to record a datapoint
    Then the API should emit a negative response with HTTP code 422
```

Die Schritte aus diesem Test sind per Parametrisierung wiederverwendbar gestaltet um in den Tests für die anderen Äquivalenzklassen erneut benutzt werden zu können. Eine vereinfachte Implementierung der Schritte könnte folgendermaßen aussehen:

```python
@given("A device {device_name} exists")
def ensure_device_exists(device_name: str):
    assert(greenhouse.has_device("lamp"))
    context.device = greenhouse.get_device("lamp")

@given("the device is of type {device_type}")
def ensure_device_is_of_type(device_type: str):
    assert(context.device.type == DeviceType.ACTUATOR)

@when("the device tries to record a datapoint")
def device_record_datapoint():
    context.response = greenhouse.record_datapoint(context.device)

@then("the API should emit a {} response with code {code}")
def api_should_respond_with(code: int):
    assert(context.response.code == code)
```

7 Zusammenfassung und Reflexion

Zusammenfassend ziehen wir eine sehr positive Bilanz aus dem Projekt. Wir haben im Laufe der Zeit viel über Softwareentwicklung und die Herausforderungen bei der Organisation dieser Entwicklung gelernt. Außerdem konnten wir einige unserer Ziele erreichen. Dazu gehört User Story 3, die Anzeige von Messdaten in Echtzeit. Die Messdaten werden automatisch ausgelesen und können überall auf einer Webseite eingesehen werden. Auf der Softwareseite ist es auch möglich, das Gewächshaus fernzusteuern (User Story 4). Dazu können über die Webseite Parameter eingestellt werden, die über das Backend an die Hardware übertragen werden.

Einige Ziele haben wir jedoch noch nicht erreicht. Wir haben noch nicht alle Sensoren und Aktoren final angeschlossen. So ist es softwareseitig schon möglich das Licht und die Bodenbewässerung anzusteuern, diese sind aber noch nicht eingebaut. Somit sind die User Stroy 1 und 2 noch nicht Final erreicht, können aber durch den Anschluss der Hardware vervollständigt werden. Dass nicht alle Ziele erreicht wurden, lag auch daran, dass nicht immer alles klappte oder auch mal falsch geplant wurde. Am Anfang haben wir gemeinsam die API Funktionalitäten besprochen und grob ausgearbeitet, jedoch waren diese noch nicht komplett ausgearbeitet, sodass im späteren Verlauf des Projektes die Teams aufeinander warten mussten, um an bestimmten Stellen weiterarbeiten zu können. Des Weiteren wurden einige wöchentliche Meetings aufgrund von Terminschwierigkeiten ausgesetzt und nicht nachgeholt, was zur Folge hatte, dass in einigen Wochen wenig bis gar kein Fortschritt erzielt wurde.
Wenn die Meetings stattfanden, waren sie durchaus produktiv. Innerhalb und zwischen den Teams wurde dann gut kommuniziert, wer was zu tun hat, was noch benötigt wird und welche neuen Aufgaben für die nächste Woche anstehen.
Insgesamt betrachten wir unser Projekt als erfolgreich. Wir haben es geschafft, unser Projekt zu planen, die Aufgaben gut zu verteilen und Ergebnisse zu erzielen. Es gab einige Fehler, die dazu geführt haben, dass wir nicht so viel erreicht haben, wie wir uns vorgenommen hatten, aber im Nachhinein haben wir diese Fehler erkannt und können sie in Zukunft vermeiden. Da das Projekt noch nicht abgeschlossen ist, können wir in Zukunft weiter daran arbeiten. So können wir sicherstellen, dass alle unsere Ziele erreicht werden. Außerdem ist während der Arbeit die Idee eines Systems mit mehreren Gewächshäusern entstanden, wofür auch schon der Grundstein gelegt wurde. Für die Zukunft kann das Projekt also noch erweitert werden.