Technische Hochschule Köln

[redacted]

Name:	Prozentuale Beteiligung:	Matrikel-nummer:	Gruppe:
[redacted]Steinker	[redacted]	[redacted]	[redacted]
[redacted] Chelly	[redacted]	[redacted]	
[redacted] Zaafouri	[redacted]	[redacted]	
[redacted]Feil	[redacted]	[redacted]	
[redacted]Buhl	[redacted]	[redacted]	
[redacted]	[redacted]	[redacted]	
[redacted]	[redacted]	[redacted]	

Thema:

Visualisierung von Maschinenampeln bei STIHL

Nummer der Abgabe:

[redacted]

Modulverantwortlicher:

[redacted]

Studiengang:

Master Maschinenbau

Semester:

[redacted]

Abgabedatum:

18.01.2025

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Beschreibung
API	Application Programming Interface
CLI	Command Line Interface
CRUD	Create-, Read-, Update-, Delete-Operationen
OEE	Overall Equipment Effectiveness
OPC-UA	OPC Unified Architecture
SQL	Structured Query Language
UI	User-Interface
URL	Uniform Resource Locator

1. Einleitung

Dieses Projekt umfasst die Neuentwicklung einer zentralen Plattform zur Echtzeit Visualisierung von Maschinenampeln bei STIHL und baut auf die Konzeptionierung aus Abgabe 1 auf [vgl.1]. Im Kontext der Industrie 4.0 ermöglicht die zentrale Visualisierung von Betriebszuständen eine datenbasierte Entscheidungsfindung. Durch eine schnelle, übersichtliche Darstellung kritischer Produktionsdaten können Mitarbeitende effizienter arbeiten. [1, S. 1] [redacted]

Das Baukastenprinzip von NodeRed ermöglicht die Einbindung der gesamten Produktionsumgebung jedoch nur auf grundlegender Ebene. Zudem ist es sehr aufwendig, neue Dashboards anzulegen und es fehlt die Skalierbarkeit der Anwendung. [2], [3]

Ziel dieses Projekts ist eine Harmonisierung der bei STIHL eingesetzten Softwarelösungen. Außerdem soll der Umgang mit der Anwendung vereinfacht werden. Auch die Skalierbarkeit auf alle STIHL-Werk soll über eine offene Architektur sichergestellt sein. Dabei soll die neue Plattform nahtlos in die vorhandene Infrastruktur integriert werden. [2]

In dieser Abgabe wird die entwickelte Lösung, einschließlich der Projektplanung, der technischen Umsetzung, der Einbindung in die bestehende Systemlandschaft sowie der Bewertung von Stärken und Schwächen, näher beschrieben.

2. Digitalisierung bei STIHL

[redacted]

[redacted]

3. Projektmethodik

In diesem Kapitel wird kurz auf das Projektmanagement eingegangen. Dies soll Transparenz in Bezug auf die methodische Vorgehensweise schaffen.

Um eine zielorientierte Bearbeitung des Projekts sicherzustellen, werden zunächst die einzelnen Aufgaben erfasst. Diese werden anschließend thematisch sortiert, zeitlich kalkuliert sowie terminiert. Das Ergebnis ist im Gantt-Diagramm (siehe Anhang 1) visualisiert. Das Vorgehen bei der Projektbearbeitung lässt sich diesem direkt entnehmen. Die einzelnen Aufgaben werden innerhalb der Projektgruppe gleichmäßig verteilt. Die eingeplanten Puffer erhöhen die Flexibilität bei unerwarteten Ereignissen und minimieren das zeitliche Risiko des Projekts. Der Abschluss wichtiger Aufgaben wird durch Meilensteine markiert. An diesen Punkten wird der Projektfortschritt reflektiert. Dabei wird auch überprüft, ob die Anforderungen des Auftraggebers und der Zeitplan eingehalten werden.

Durch ein wöchentliches Regelmeeting der Projektgruppe (mittwochs) wird kontinuierlich die Weiterentwicklung des Projekts sowie die engmaschige Überprüfung des Zeitplans gewährleistet. Im zweiwöchigen Rhythmus ist STIHL als Auftraggeber ebenfalls Teilnehmer dieses Regelmeetings. Bei diesen Terminen werden der Projektfortschritt und das weitere Vorgehen besprochen. Zu jedem Meeting wird zur Dokumentation ein Besprechungsprotokoll angelegt (siehe Anhang 4).

4. Detaillierte Ausarbeitung und technische Dokumentation

Ziel des Kapitels ist es, die Implementierung und technischen Details des Projekts transparent und nachvollziehbar zu dokumentieren. Die Implementierung und Architektur basiert auf dem Vorkonzept B aus Abgabe 1 [vgl. 1, S. 10-12].

Die Dokumentation erfolgt vom Allgemeinen zum Spezifischen und getrennt nach Front- und Backend. Es wird dabei auf die Anwendung, die grundlegenden Funktionen und die wesentlichen Codeblöcke der Software eingegangen. Da die Angabe des gesamten Codes aufgrund der komplexen Dateizusammenhänge und dem hohen Codeumfang in diesem Dokument nicht sinnvoll möglich ist, erfolgt die Beschreibung durch im Text eingebettete Codeausschnitte. Diese detaillierte, deskriptive Code-Dokumentation ist eine Forderung von STIHL, um die erarbeite Lösung tiefgehend nachvollziehen zu können. Dies soll es ermöglichen, die Software weiterzuentwickeln und für den Einsatz in allen Werken anpassen zu können. [6]

Der vollständige Code ist im GitHub-Repository *fynnbuhl/Maschinenampel* (siehe: https://github.com/fynnbuhl/Maschinenampel/) verfügbar.

In der Tabelle 4-1 sind die verwendeten Frameworks und deren Versionen aufgeführt. Diese Auswahl wird von STIHL in der Projektbeschreibung vorgegeben, um die Skalierbarkeit für alle Werke sicherzustellen. [2]

Tabelle 4-1: Verwendete Frameworks

Frameworks	Version	Beschreibung
Angular CLI	17.3.10	Plattform zur Entwicklung clientseitiger Single-Page-Anwendungen [7]
Node.js	18.20.4	JavaScript Laufzeitumgebung [8]
ASP.NET Core	.NET 8.0	Als Backend (API) fungierendes Web-Framework [9]

4.1 Architekturübersicht

Die Abbildung 4-1 veranschaulicht die Interaktion der wesentlichen Architekturbestandteile.

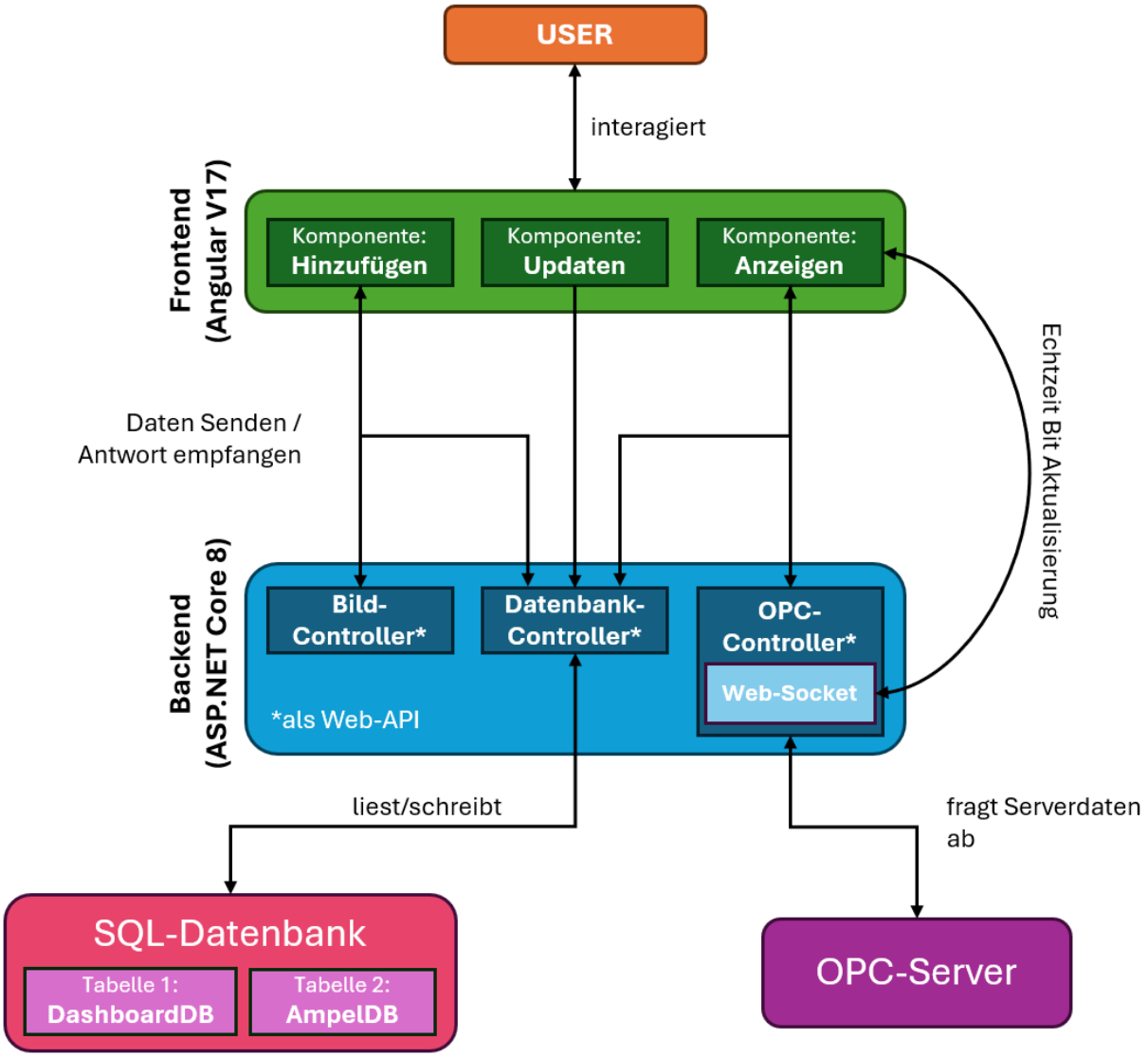


Abbildung 4-1: Architekturübersicht

Der Nutzer (User) interagiert mit der Software über das Frontend. Dabei stehen drei Komponenten zur Verfügung. Jede der drei Komponenten erfüllt einen Teilbereich des in Abgabe 1 genannten Anforderungsprofils [vgl. 1, Kapitel 4.1]. Die einzelnen Funktionen werden in Kapitel 4.2 beschrieben. Allgemein gilt: Das Frontend sendet Daten des Nutzers an die jeweiligen Web-API-Controller des Backends. Die Controller verarbeiten diese und führen die notwendigen SQL- bzw. OPC-Serverabfragen aus. Die entsprechende Antwort wird zurück an den Client gesendet und dort visualisiert.

Der OPC-Controller stellt zusätzlich einen Web-Socket bereit, damit die Bits zum Visualisieren der Ampelzustände kontinuierlich dem Frontend bereitgestellt werden können.

Die Architektur dieses Projekts ist so gestaltet, dass Frontend, Backend und Datenbank klar voneinander getrennt sind. Dies ermöglicht eine skalierbare und wartungsfreundliche Umsetzung. Außerdem wird die Integration in die bestehende Produktionsumgebung von STIHL erleichtert [7], [9].

4.2 Frontend

Das Frontend ist mit Angular entwickelt. Die Architektur des Clients folgt einer komponentenbasierten Struktur. Dies stellt eine klare Trennung von Logik und Darstellung sicher. In den folgenden Abschnitten wird auf die wichtigsten Komponenten, Services und das Routing eingegangen, um die grundlegende Funktionalität des Frontends zu verdeutlichen. Dabei visualisiert Abbildung 4-2 die wichtigsten Bestandteile der Verzeichnisstruktur im Frontend.

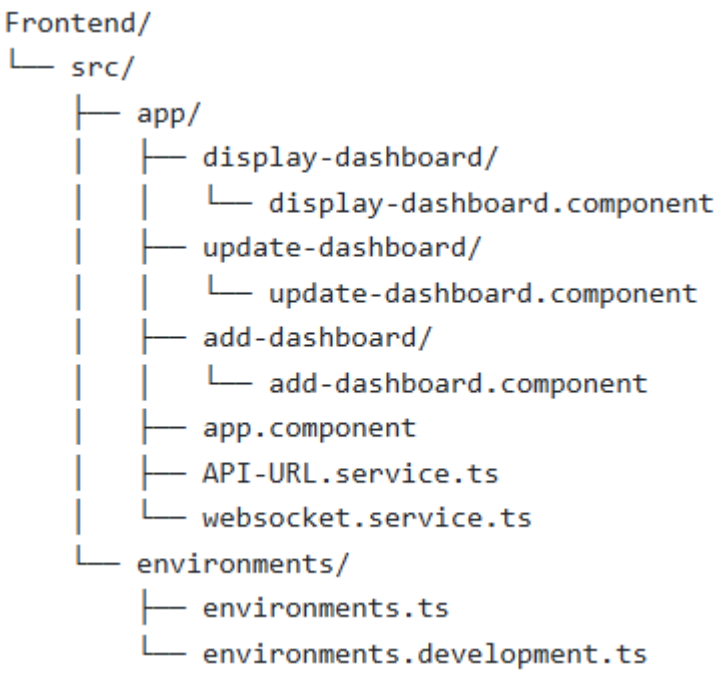


Abbildung 4-2: Verzeichnisstruktur vom Frontend

Die Komponente *app.component* stellt die root-Komponente des Projekts dar und wird bei jedem Programmstart automatisch aufgerufen. Aus diesem Grund stellt diese Komponente das Menü der Anwendung bereit (siehe Abbildung 4-3). Für jeden Menü-Button ist ein *routerLink*-Attribut hinterlegt, welches dem Nutzer beim Klicken die jeweilige Sub-Komponente (**-dahboard.component*) einblendet [10]. Durch die Bedingung `<div *ngIf="router.url === '/'">` in der root-Komponente wird das Menü

ausgeblendet, sobald ein Routing zu einer anderen Komponente ausgewählt wird. Die einzelnen Kernfunktionen sind in die jeweilige Sub-Komponente ausgelagert, um auch bei einer Funktionserweiterung eine konsistente Menüführung zu ermöglichen und die Anwendung leichter an bestehende Tools bei STIHL anzupassen [11].

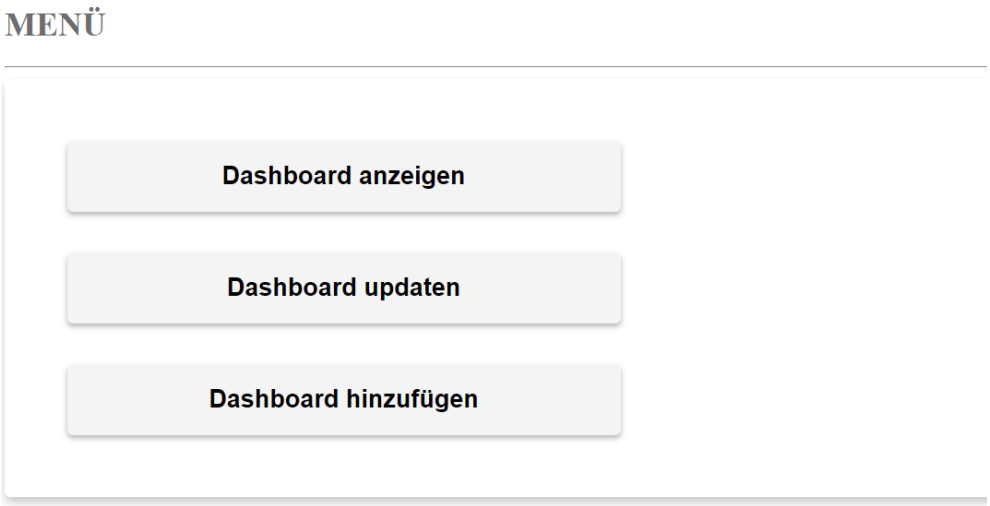


Abbildung 4-3: UI Hauptmenü

Der Service *API-URL.service* wird von allen Sub-Komponenten benötigt. Dieser stellt die Serveradressen der verschiedenen Web-API-Controller (vgl. Abbildung 4-1) bereit. Die ausgelagerte, separate Initialisierung der URL-Variablen ermöglicht es, die Adressen unkompliziert anzupassen und vermeidet eine redundante Definition in allen Komponenten.

Der Service *websocket.service* deklariert einen Web-Socket. Die Komponente *display-dashboard.component* ruft diesen auf, um kontinuierlich mit dem Backend zu kommunizieren. Die ausgelagerte Definition erhöht die Codelesbarkeit und reduziert den Wartungsaufwand innerhalb der Komponente.

Der Ordner *Environments* enthält verschiedene Konfigurationen für unterschiedliche Umgebungen. Die Standardumgebung ist die Produktion (*environments.ts*). Zum Entwickeln wird die Umgebung Development (*environments.development.ts*) genutzt. Dies ermöglicht es, Umgebungsabhängigkeiten flexibel zu handhaben, ohne den Code anzupassen. Die entsprechende Umgebung kann durch ein Flag beim Erstellen festgelegt werden. [12]

4.2.1 Dashboard anzeigen

Klickt der Nutzer im Menü auf „Dashboard anzeigen“ wird `routerLink="/display-Dashboard"` gesetzt. Die dadurch eingeblendete Oberfläche besteht aus den zwei wesentlichen und in Abbildung 4-4 dargestellten Blöcken.

Abbildung 4-4: Funktionsweise des Menüs zur Dashboard Auswahl

Die Sichtbarkeit dieser Blöcke wird über die Variabel `selectedID` gesteuert. Standardmäßig hat diese Variable den Wert 0. Durch die Bedingungen in den jeweiligen div-Containern wird deswegen nur das in Orange umrahmte Sub-Menü angezeigt. Hier werden alle in der Datenbank gespeicherten Dashboards zeilenweise aufgelistet. Die zugehörigen Daten werden in der Antwort des http GET-Requests des API-Endpunkts *api/DBController/getDashboards* bereitgestellt. Wenn der Nutzer ein Dashboard im Sub-Menü auswählt, wird die Methode `viewBoard()` aufgerufen und die Variable `selectedID` auf die eindeutige ID des gewählten Dashboards gesetzt. Durch diese Änderung ist die Bedingung des ersten Blocks (orange) nicht mehr erfüllt, sodass das Sub-Menü ausgeblendet wird. Gleichzeitig wird die Bedingung des zweiten Blocks (grün) erfüllt. Infolgedessen wird das entsprechende Dashboard, wie auf Abbildung 4-5 dargestellt, eingeblendet.

Der „Menü“-Button setzt `routerLink="/"` und der User wird zurück zum Hauptmenü geleitet. Der „Ansicht wechseln“-Button ruft die Methode `clearSelection()` auf und setzt dadurch die Variablen der aktuellen Auswahl (siehe z. B. `selectedID`) auf ihre Standardwerte (*null bzw. 0*) zurück. Dem Nutzer wird wieder das Sub-Menü (orange) angezeigt.

Abbildung 4-5: Visualisierung eines Dashboards [Hintergrund aus 13]

Die Methode `viewBoard()` ruft außerdem `getAmpelnVonBoard()` asynchron auf. Diese Methode sendet einen http GET-Request an den API-Endpunkt *api/DBController/getAmpeln* und übermittelt in der URL die `selectedID` an den Datenbank-Controller. Sobald das Backend die Antwort gesendet hat, iteriert der Client zeilenweise über alle erhaltenden Ampeln. Zunächst wird der Antwort-String `COLORS` in ein zweidimensionales Array zerlegt und in `colorsArray: string[][]` gespeichert. Jede Zeile dieses Arrays repräsentiert dabei eine Ampel und jede Spalte ein Licht der jeweiligen Ampel. Anschließend wird der Antwort-String `OPC_TagList` verarbeitet. Hierbei wird jedem Element der Antwort-String `OPC_Addr` als Präfix hinzugefügt und mit einem Punkt getrennt. Daraus resultiert das zweidimensionale Array `OPC_AddArray: string[][]`. Es enthält den vollständigen Adressstring für alle OPC-Nodes der angefragten Ampeln (vgl. Abbildung 4-12). Dieses Array wird als JSON-String mit der

URL des Web-Sockets zur Variable *fullUrl* zusammengesetzt (`fullUrl = ` `${url}?addresses=${AddrData}` `;`) und dadurch bei einer Websocket-Verbindung an den OPC-Controller übergeben. Dieser Controller stellt dem Client die OPC-Bits in Echtzeit bereit (vgl. Abbildung 4-13). Die eingehenden Daten werden im `OPC_BITArray` gespeichert und dadurch der aktuelle Ampelstatus visualisiert.

Der Hintergrund wird aus dem gespeicherten Bild-Pfad aus *wwwroot* vom Backend geladen. Anschließend passt die Methode `fitScreenAspectRatio()` die Darstellung des Dashboards an das aktuelle Seitenverhältnis des Browserfensters an. Dadurch wird sichergestellt, dass das Layout unabhängig von der Bildschirmgröße immer optimal dargestellt wird.

Das User-Interface visualisiert die Ampeln und ihre aktuellen Zustände in einem dynamisch generierten Layout. Wie in Abbildung 4-6 dargestellt, wird hierzu die **ngFor*-Direktive verwendet, wobei für jede Ampel eine HTML-Struktur der Klasse `ampel` erzeugt wird. Der Stil dieser Klasse wird durch die Methode `getElementStyles()` an jede Ampel individuell angepasst.

```
…
    <div class="dashboard-container" [style.aspect-ratio]="aspectRatio">

      <!-- Hintergrundbild des Layouts -->
      <div class="background-container">
        <!-- Anzeige des ausgewählten Hintergrundbildes -->
        <img class="hallenlayout_img" src={{selectedIMG}} />
      </div>

        <!-- Schleife durch die Ampel-Liste und zeige für jedes Element die
Ampel an -->
        <div *ngFor="let ampel of Ampeln; let ampIndex = index" class="ampel"
[ngStyle]="getElementStyles(ampel)">
          <!-- Iteriere über die Farben der jeweiligen Ampel -->
          <!-- Die Ampel-Farben werden aus dem colorsArray gezogen -->
          <div *ngFor="let color of colorsArray[ampIndex]; let i = index"
class="circle-wrapper">
            <!-- Zeichne den Kreis: Wenn der zugehörige Bit-Wert 1 ist, wird
der Kreis in der jeweiligen Farbe angezeigt, andernfalls schwarz -->
            <div class="circle"
                 [ngStyle]="{'background-color': OPC_BITArray[ampIndex][i] ===
1 ? color : 'black'}">
            </div>
          </div>
          <!-- Anzeige der Ampel-ID -->
          ID:{{ampel.ID}}
        </div>

    </div>

…
```

Abbildung 4-6: Codeausschnitt zur Visualisierung einzelner Ampeln

Innerhalb jeder Ampel-Komponente wird über die einzelnen Farben (`colorsArray`) iteriert, wobei der `ampIndex` zur Zuordnung verwendet wird. Jedes Farbelement wird in einem eigenen div-Container als Farbkreis dargestellt. Die Hintergrundfarbe jedes Kreises wird durch das *ngStyle*-Attribut ebenfalls dynamisch gesetzt: Ist der zugehörige Wert im `OPC_BITArray` gleich 1, wird der Kreis in der definierten Farbe angezeigt; andernfalls bleibt er schwarz. Angular aktualisiert die Darstellung automatisch, sobald der Web-Socket neue Daten in das `OPC_BITArray` sendet.

4.2.2 Dashboard updaten

Klickt der Nutzer im Menü auf „Dashboard updaten“ wird `routerLink="/update-Dashboard"` gesetzt. Die Funktionalität der Auswahl des zu bearbeitenden Dashboards erfolgt identisch, wie oben für die Komponente *Dashboard anzeigen* beschrieben. Das User-Interface dieser Komponente wird wie auf Abbildung 4-7 dargestellt eingeblendet.

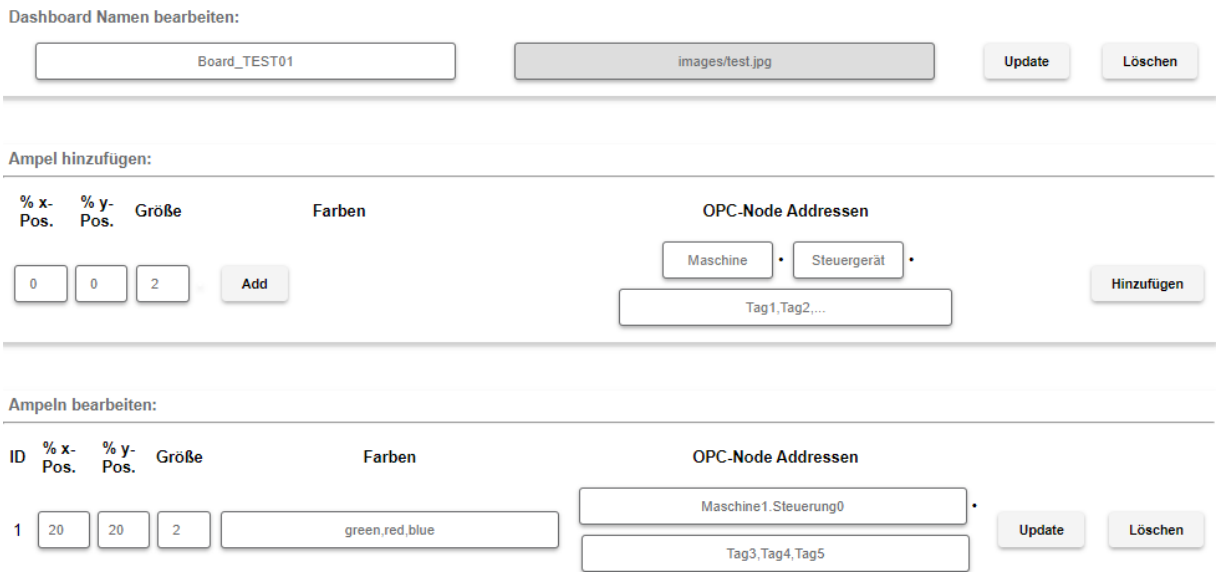


Abbildung 4-7: UI der Komponente Dashboard updaten

In dieser Komponente können drei Aufgaben erledigt werden. Diese sind: (A) Dashboard Namen bearbeiten, (B) Ampel hinzufügen und (C) vorhandene Ampeln bearbeiten.

(A): Um den Namen zu bearbeiten, kann im Textfeld der neue Name eingetragen und über den Button „Update“ aktualisiert werden. Der Button ruft über die Methode `updateBoard(ID: number)` per http POST-Request den API-Endpunkt *api/DBController/updateDashboardName* des Datenbank-Controllers mit dem neuen Namen im JSON-Body auf. Als ID wird die `selectedID` des ausgewählten Dashboards übergeben. Die Aktualisierung des Hallenlayouts ist nicht vorgesehen, da dies eine Neupositionierung aller Ampeln erfordern würde. Stattdessen kann einfach das jeweilige Dashboard gelöscht und durch ein neues ersetzt werden. Der „Löschen“-Button führt die Methode `deleteBoard(ID: number)` aus. Per http POST-Request wird der API-Endpunkt *api/DBController/deleteDashboard* aufgerufen und das ausgewählte Dashboard inklusive aller Ampeln aus der Datenbank gelöscht.

(B): Um eine neue Ampel dem Dashboard hinzuzufügen, muss die Eingabemaske ausgefüllt werden. Dabei können maximal sechs Farben ausgewählt werden (Kundenvorgabe [1, S. 8]). Eine doppelte Auswahl ist nicht zulässig, da dies der Norm DIN EN 60073 widerspricht [14]. Die OPC-Tags müssen mit Kommata voneinander getrennt werden. Die Dashboard-ID wird automatisch verknüpft, um Fehler bei der Eingabe zu verhindern. Der „Hinzufügen“-Button führt die Methode `addAmpel(ID: number)` aus. Als ID wird die `selectedID` des ausgewählten Dashboards übergeben und ein http POST-Request an den API-Endpunkt *api/DBController/addAmpel* gesendet. Ein beispielhafter Body ist in Abbildung 4-8 dargestellt.

```
{
  "Dashboard_ID": 1,
  "POS_X": 50,
  "POS_Y": 20,
  "SIZE": 3,
  "ColorCount": 3,
  "COLORS": "red,green,blue",
  "OPC_Addr": "Maschine01.Steuer01",
  "OPC_TagList": "Tag1,Tag2,Tag3"
}
```

Abbildung 4-8: Beispielhafter JSON-Body einer API-Anfrage zum hinzufügen einer Ampel

(C): In diesem Abschnitt sind alle dem aktuellen Dashboard zugeordneten Ampeln aufgelistet. Über die Eingabemaske können ihre Eigenschaften verändert werden. Die einzelnen Farben müssen dabei mit Kommata getrennt und als gültige CSS-Farbe (Name, HEX, RGB oder HSL) angegeben werden. Die OPC-Adresse muss mit Punkten, die OPC-Tag-Liste mit Kommata getrennt werden. Leerzeichen dürfen nicht verwendet werden. Die Buttons „Update“ und „Löschen“ funktionieren analog zu der Beschreibung in (A). Als ID wird die jeweilige Ampel-ID übergeben. Die API-Endpunkte lauten *api/DBController/updateAmpel* und *api/DBController/deleteAmpel*.

4.2.3 Dashboard hinzufügen

Klickt der Nutzer im Menü auf „Dashboard hinzufügen“ wird `routerLink="/add-Dashboard"` gesetzt. Das User-Interface dieser Komponente wird wie auf Abbildung 4-9 dargestellt eingeblendet.

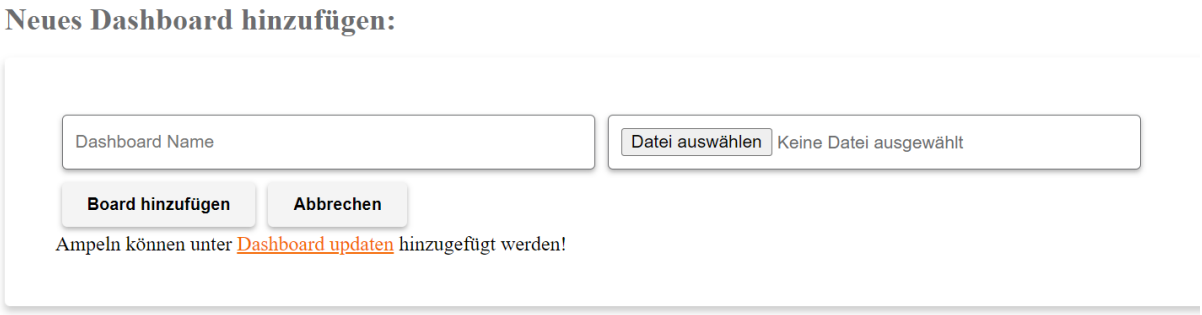


Abbildung 4-9: UI der Komponente Dashboard hinzufügen

Hier kann der Anwender den neuen Dashboard-Namen eintragen und ein Bild (*.png, *.jpg, *.jpeg) des Hallenlayouts auswählen. Die Dateibeschränkung wird getroffen, um den Implementierungsaufwand zu reduzieren. Bei einem Klick auf „Board hinzufügen“ wird die Methode `add_board()` aufgerufen und der „Hinzufügen“-Button für die Laufzeit der Methode deaktiviert. Anschließend wird die Methode `uploadFile()` asynchron aufgerufen, um das Bild auf dem Backend-Server zu speichern. Dazu wird die Datei per http POST-Request an den API-Endpunkt des Bild-Upload-Controllers gesendet. Wenn der Speicherpfad und das Seitenverhältnis als Antwort vom Server erhalten wurden, wird die Methode `saveToDB()` aufgerufen. Diese ruft ebenfalls per http POST-Request den API-Endpunkt *api/DBController/addDashboard* des Daten-

bank-Controllers auf und übergibt dabei den Dashboard-Namen sowie den Speicherpfad und das Seitenverhältnis des Hintergrundbildes als Body im JSON-Format. Sobald die Daten erfolgreich in der Datenbank gespeichert sind, wird die Komponente neu geladen, um die Eingabefelder zurückzusetzen und der „Board hinzufügen“-Button wird wieder freigegeben. Der Nutzer wird über den erfolgten Vorgang informiert. Der „Abbrechen“-Button lädt ebenfalls die Komponente neu, um die Eingabefelder zurückzusetzen.

4.3 Backend

Das Backend des Projekts ist mit ASP.NET Core entwickelt und dient als zentrale Schnittstelle zwischen dem Frontend, der Datenbank und dem OPC-UA Server. Es stellt die notwendigen APIs bereit. Im Folgenden werden die Controller sowie deren Interaktion mit der SQL-Datenbank und dem OPC-UA-Server beschrieben. Dabei visualisiert Abbildung 4-10 die wichtigsten Bestandteile der Verzeichnisstruktur im Backend.

```
Backend/
├── Pakete/
│   ├── SQLServer      (V8.0.10)
│   ├── Tools          (V8.0.10)
│   ├── Opc.Ua.Client  (V1.5.374.126)
│   ├── SikaSharp      (V2.88.9)
│   └── ...
├── wwwroot/
│   └── images/
│       └── Hallenlayout01.png
├── Controllers/
│   ├── imgUpload_Controller
│   ├── DB_Controller
│   └── OPC_Controller
├── Services/
│   └── OPC_Service
├── Datenbank/
│   ├── Database.mdf
│   ├── SQLQuery_AmpelDB.sql
│   └── SQLQuery_DashboardDB.sql
├── appsettings.json
└── Program.cs
```

Abbildung 4-10: Verzeichnisstruktur vom Backend

Zentraler Bestandteil ist die Datei *Program.cs*, die den Einstiegspunkt der Anwendung darstellt. Sie konfiguriert den Webserver und registriert die notwendigen Middleware-Komponenten.

Im Ordner *Pakete* befinden sich alle verwendeten NuGet-Pakete. Sie sind essenziell für die Funktionalität der Anwendung. Alle verwendeten Pakete sind Open-Source und MIT-Lizensiert. Dies ist eine Anforderung von STIHL. [2]

Die zentralen Pakete der Anwendung sind:

- *Microsoft.EntityFrameworkCore.**SqlServer*:** ermöglicht die Anbindung an die SQL-Datenbank [15].
- *Microsoft.EntityFrameworkCore.**Tools*:** bietet Tools für die Datenbankmigration und -verwaltung [16].
- *OPCFoundation.NetStandard.**Opc.Ua.Client*:** wird für die Kommunikation mit dem OPC-UA-Server verwendet [17].
- ***SikaSharp*:** wird für die Verarbeitung und Speicherung von Bildmetadateien verwendet [18].
- *Microsoft.*AspNetCore.Mvc.**NewtonsoftJson:** wird für die Serialisierung und Deserialisierung von JSON-Daten verwendet [19].

Der Datenbank-Ordner enthält alle notwendigen Ressourcen zur Verwaltung einer lokalen Datenbank. Dazu gehören sowohl die Initialisierungsskripte (*SQLQuery_<Tabellenname>.sql*) für die Tabellen *DashboardDB* und *AmpelDB*, als auch die eigentliche Datenbank *Database.mdf*. Die Struktur und die Tabellenbeziehung der Datenbank werden in Kapitel 4.4 ausführlich beschrieben. Die Verbindungszeichenfolge zu dieser Datenbank ist in der Datei *appsettings.json* gespeichert (siehe: `"ConnectionStrings": {"DBConnect": "…"}`).

Im Ordner *Controller* sind alle Web-API-Controller der Anwendung gespeichert. Auf die einzelnen Controller wird in den folgenden Kapiteln separat eingegangen.

4.3.1 Datenbank-Controller

Der Datenbank-Controller *DB_Controller*.cs dient als API-Schnittstelle für die Verwaltung und Abfrage von Daten der SQL-Datenbank. Dieser implementiert CRUD-Operationen (Create, Read, Update, Delete) für die beiden Entitäten Dashboards und Ampeln. Die Ergebnisse werden in typisierte Datenmodelle überführt und als standardisierte Antwort an den Client zurückgegeben, wobei asynchrone Datenbankzugriffe für optimale Performance verwendet werden [20]. Die Endpunkte der verwendeten CRUD-Operationen sind im Anhang 2 aufgelistet.

4.3.2 Bild-Upload-Controller

Der Bild-Upload-Controller *imgUploadController.cs* speichert die hochgeladenen Bilddateien. Die API ist über den Endpunkt *api/imgUpload/upload* erreichbar. Die entgegengenommenen Dateien werden im Verzeichnis *wwwroot/images* gespeichert. Dieses Verzeichnis ist das Stammverzeichnis für statische Inhalte und verbessert die Sicherheit und Wartbarkeit der Anwendung [21]. Zur Vermeidung von Namenskonflikten wird der Dateiname durch einen eindeutigen Zeitstempel ergänzt. Zusätzlich werden mithilfe der Bibliothek SikaSharp die Bildmetadaten Breite und Höhe ausgelesen, um daraus das Seitenverhältnis zu berechnen. Bei Erfolg sendet der Controller den relativen Speicherpfad des Hallenlayouts und das zugehörige Seitenverhältnis im JSON-Format zurück an den Client (vgl. Abbildung 4-11).

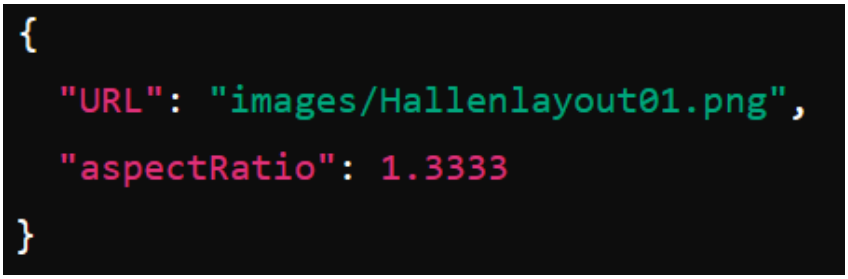


Abbildung 4-11: Beispiel Antwort des Bild-Upload-Controllers

4.3.3 OPC-Controller

Der OPC-Controller *OPC_Controller.cs* bildet die Schnittstelle zum OPC-UA Server des Kunden. Über den Endpunkt *api/OPCController/connectWebSocket* kann der Client eine Web-Socket-Verbindung mit dem Controller aufbauen. Dieser Web-Socket ermöglicht den Echtzeit Datenaustausch zwischen Front- und Backend.

Gleichzeitig werden über die Verbindungs-URL auch die Adressstrings der zu visualisieren Ampeln (OPC-Nodes) an den Controller übergeben. Diese werden deserialisiert und im zweidimensionalen Adressarray `string[][] OPC_AddrArray` gespeichert. Ein beispielhafter Inhalt des Adressarrays ist in Abbildung 4-12 dargestellt.

```
["Maschine1.Steuerung1.Tag1", "Maschine1.Steuerung1.Tag2"],
["Maschine2.Steuerung2.Tag1", "Maschine2.Steuerung2.Tag2", "Maschine2.Steuerung2.Tag3"]
```

Abbildung 4-12: Beispielhaftes Adressarray mit den Node-Adressen der abzufragenden Ampeln

Anschließend wird die Struktur diese Arrays kopiert und aus der Kopie das zweidimensionale Bit-Array `int[][] OPC_BitArray` mit dem Wert *-1* initialisiert. Da *-1* kein gültiger Bitzustand ist, können Fehler bei der OPC-Server-Abfrage schneller erkannt werden.

Den Adressstring und die Verbindungs-URL zusammen an den Websocket zusenden wird statt eines dedizierten API-Endpunkts gewählt, um Multi-Client-Sitzungen zu ermöglichen. Da der Controller (inklusive der verwendeten Variablen) eine transiente Lebensdauer hat und bei jedem Clientaufruf eine neue Instanz erstellt werden würde. Daher wären die von einem Endpunkt initialisierten Adressen nicht vom anderen Endpunkt verwendbar. Eine Variablendeklaration als *static* würde verhindern, dass unterschiedliche Clients auf die Anwendung zugreifen. [22]

Beim Programmstart wird eine Verbindung zum OPC-UA-Server hergestellt. Dazu wird eine Instanz des *OPC_Service* als *Singleton-HostedService* initialisiert. Dieser Service verbindet sich *anonym* und ohne Sicherheitsrichtlinie mit dem OPC-UA Server. Der OPC-Server muss dazu entsprechend konfiguriert werden (Anleitung siehe Kapitel

5.1). In Absprache mit STIHL wird eine *UserIdentity* und/oder ein *Sicherheitszertifikat* erst bei Bedarf, eigenständig im Service ergänzt [6].

Der aktive Web-Socket ruft alle 1000 Millisekunden (in *appsettings.json* einstellbar) die Controller-Methode `UpdateBits(OPC_AddrArray, OPC_BitArray)` auf. Diese iteriert über alle Elemente des Adressarrays und fragt mithilfe der Methode `ReadNodesAsync()` des OPC-Services den Bitzustand der jeweiligen Ampel vom OPC-Server ab. Die Zustände werden im Bit-Array aktualisiert. Sobald die neuen Daten vollständig zur Verfügung stehen, sendet der Web-Socket das Bit-Array zurück an den Client (vgl. Abbildung 4-13).

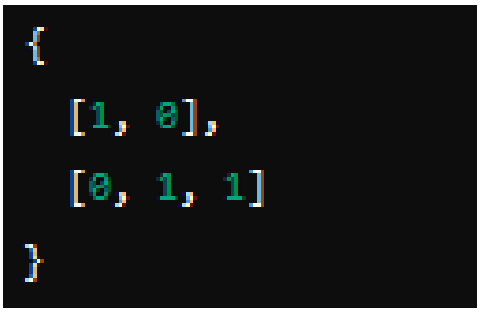


Abbildung 4-13: Beispielhafter JSON-Body der abgefragten Bit-Zustände

Enthält die Antwort Elemente mit dem Debugging-Wert -2, ist entweder keine Verbindung mit dem OPC-Server möglich oder einer der abgefragten Nodes existiert nicht bzw. ist nicht vom Typ *Boolean*. Im Backend-Terminal wird dann die spezifische Ursache angegeben.

Das ausschließliche Übergeben der Ampel-Adressen des aktuellen Dashboards und das selektive Abfragen der zugehörigen OPC-Bits erhöht zwar den Rechenaufwand der Controllerseite, sorgt aber dafür, dass der Client entlastet wird. Der Client müsste andernfalls die Bits aller auf dem OPC-Server vorhandenen Nodes selektieren und mit den Ampeln verknüpfen. Die Rechenleistung des Backend-Servers ist jedoch höher als die der clientseitigen Mini-PCs [3]. Um die vorhandenen Ressourcen effizient zu nutzen, wird daher der oben beschriebene Ansatz gewählt.

4.4 SQL-Datenbank

Zur Speicherung und Verwaltung der für die Anwendung relevanten Daten wird eine SQL-Datenbank eingesetzt. Diese gewährleistet eine strukturierte und effiziente Speicherung von Informationen [23]. Die verwendete Datenbank umfasst die zwei Tabellen:

- **DashboardDB:** enthält die grundlegenden Informationen der Dashboards (Name, Pfad zum Hintergrundbild, Seitenverhältnis des Bildes)
- **AmpelDB:** enthält die spezifischen Daten der digitalen Maschinenampeln (x-Pos., y-Pos., Größe, Farbanzahl, Farbliste, OPC-Node-Adresse, OPC-Taglist)

Die Beziehungen der Entitäten *Dashboards* und *Ampeln* sind in Abbildung 4-14 dargestellt.

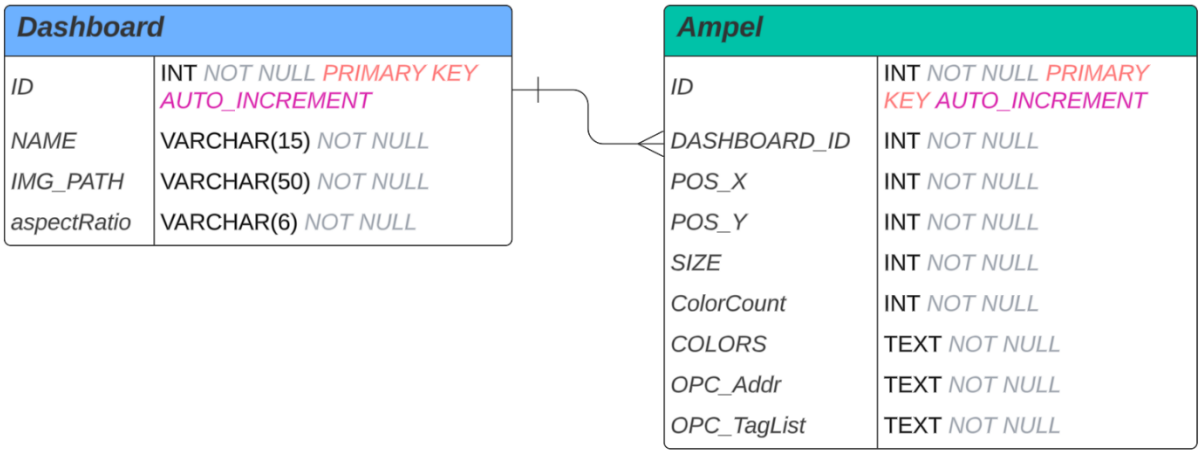


Abbildung 4-14: Entity-Relationship-Modell der SQL-Datenbank

Für das vorliegende Projekt wird eine lokale SQL-Datenbank verwendet, es besteht auch die Möglichkeit, einen externen SQL-Server zu integrieren. Dazu muss die Verbindungszeichenfolge angepasst werden (siehe Kapitel 5.1). Der lokale Ansatz wird gewählt, um die Komplexität so gering wie möglich zu halten. Dadurch kann das Testen vereinfacht und Cloud-Kosten gespart werden. [24]

5. Integration und Anwendung bei STIHL

Dieses Kapitel beschreibt die notwendigen Schritte für die erfolgreiche Integration der Softwarelösung in die Produktionsumgebung von STIHL. Besonderes Augenmerk liegt dabei auf den technischen Schnittstellen und den Konfigurationseinstellungen der bestehenden Infrastruktur. Dieses Kapitel wird nach Aufforderung von STIHL aufgenommen, um einen praxisnahen Leitfaden bereitzustellen. [25]
Es wird dabei vorausgesetzt, dass die in [1] beschriebenen Rahmenbedingungen (Kapitel 3.2) und Annahmen (Kapitel 3.3) durch STIHL erfüllt sind.

5.1 Integration

Zuerst muss das Projekt von GitHub (https://github.com/fynnbuhl/Maschinenampel) geklont und der Visual Studio Projektordner in einem dedizierten Pfad gespeichert werden. Um die Kommunikation des Front- und Backends zu ermöglichen, muss ggf. die Server-URL inklusive Port des Backend-Servers in den jeweiligen Environments des Frontends aktualisiert werden (siehe: `serverUrl: '<BackendServerURL>:<PORT>'`).

Anschließend ist der OPC-UA Emulator KEPServer zu konfigurieren:

- **KEPserver Configuration**: *Bearbeiten->Eigenschaften->OPC UA* „Client-Sitzungen: Anonyme Anmeldung zulassen = ja“ (vgl. Abbildung 5-1)

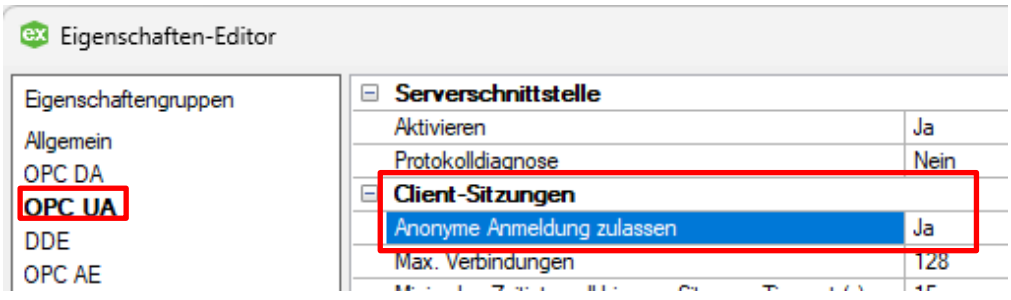


Abbildung 5-1: Abbildung KEPServer Eigenschaften-Editor

- **KEPserver OPC UA Configuration Manager**: „Sicherheitsrichtline: keine“ aktivieren (vgl. Abbildung 5-2)

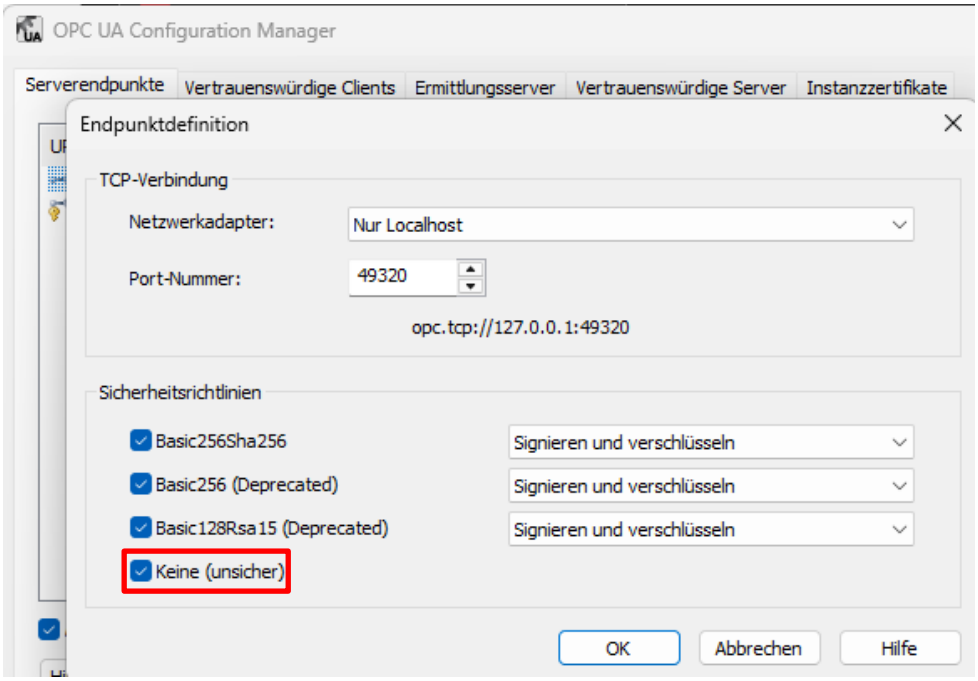


Abbildung 5-2: Configuration Manager - Endpunktdefinition

Die verwendeten OPC-UA Serverdaten (URL, UpdateIntervall_ms, etc.) müssen in den App-Settings aktualisiert werden (siehe: *Backend/appsettings.json*). Bei Verwendung einer externen SQL-Datenbank muss diese vorbereitet und eingebungen werden:

- Die Tabellen *DashboardDB* und *AmpelDB* mit der jeweiligen SQL-Query (siehe: *Backend/Datenbank/SQLQuery_<Tabellenname>.sql*) erstellen.
- Die Verbindungszeichenfolge der zu verwendenden SQL-Datenbank in den App-Settings aktualisieren (siehe: *Backend/appsettings.json:* `"ConnectionStrings": {"DBConnect": "<Verbindungszeichenfolge>"}`).

Alternativ kann die im Projekt enthaltene, lokale Datenbank verwendet werden. Dazu bleibt die Verbindungszeichenfolge unverändert.

Zum Testen der Funktionalität ist der Front- und Backendserver zu starten und der Client im Browser aufzurufen. Währenddessen können die Debugging-Informationen der Konsolen überwacht werden.

5.2 Troubleshooting

Die Konsolen im Browser und auf dem Server zeigen Informationen über die laufenden Prozesse an. Diese Nachrichten sind wichtig, um das System zu überwachen. Wenn es Fehler oder Probleme bei der Projektbereitstellung gibt, helfen die Nachrichten dabei, die Ursache zu isolieren. Im Folgenden sind zwei häufige Fehler dargestellt.

Die Ampeln bleiben dunkel:

- Sicherstellen, dass der OPC-Server aktiv und richtig konfiguriert ist (vgl. Kapitel 5.1).
- Durch die Statusmitteilung der Backend-Konsole die erfolgreiche Verbindung mit dem OPC-Server sicherstellen.
- Korrektheit des OPC_Bitarray überprüfen (vgl. Kapitel 4.3.3).

Der SPA Server kann nicht gestartet werden:
(In Zusammenhang mit *npm start* und der Fehlermeldung: *„There was an error exporting the HTTPS developer certificate to a file.“*)

- Sicherstellen, dass die Versionen der verwendeten Frameworks mit den in diesem Projekt verwendeten Versionen übereinstimmen (vgl. Tabelle 4-1).
    - Ab .NET Version 9 wird das Verzeichnis, in welches das Zertifikat exportiert wird, nicht mehr erstellt, wenn dieses Verzeichnis nicht existiert [26].
- Das Verzeichnis *%APPDATA%\ASP.NET\https* prüfen und ggf. manuell erstellen.
- Anschießend die alten Zertifikate löschen und neu generieren [26]:

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

6. SWOT-Analyse

Im folgenden Abschnitt wird eine SWOT-Analyse für die Softwarelösung „Visualisierung von Maschinenampeln“ durchgeführt. Dadurch sollen die Stärken und Schwächen der Lösung aufgedeckt und Chancen beziehungsweise Risiken für STIHL gefunden werden [27]. Die dazugehörige Tabelle ist im Anhang 3 dargestellt. Im Anschluss werden auf Grundlage der SWOT-Analyse Handlungsempfehlungen für die Implementierung bei STIHL formuliert.

6.1 Stärken

Die Stärke von Maschinenampeln ist die Verbesserung der Transparenz durch die Abrufbarkeit des Maschinenstatus über visuelle Signale in der Produktion [28]. Hinzu kommt, dass diese Signale auch an anderen Orten abgerufen werden können. Durch die standardisierten Ampelsignale wird eine effiziente Kommunikation gefördert, weil jeder Mitarbeitende die Bedeutung der Farben kennt [14]. Mithilfe der proaktiven Visualisierung der entwickelten Anwendung können Störungen an den Maschinen in Echtzeit erkannt und Stillstandszeiten reduziert werden. Die Softwarearchitektur des Konzepts ermöglicht zudem eine Anpassung an die verschiedenen Maschinen und Produktionslinien.

Eine weitere Stärke ist die Harmonisierung der unterschiedlichen Programme von STIHL. Dies wird erreicht, indem die Softwarelösung auf den Standard-Software-Stack von STIHL aufbaut [2]. Der Aufbau der Software ist für jeden Standort weltweit gleich, weswegen diese Lösung überall implementiert werden kann. So kann eine Synergie zwischen den Standorten geschaffen werden.

Durch das Nutzen der vorhandenen Infrastruktur (bestehende Maschinenampeln, bereits installierte Visualisierungsmöglichkeiten, Mini-PCs, etc.) werden die anfallenden Kosten und die Zeit zur Integration minimiert. Die Software wird ausschließlich auf internen Servern betrieben und gibt somit keine Daten an andere Unternehmen weiter. Durch die „offene“ Software hat STIHL zudem die Möglichkeit, den Programmcode und die Funktionalität der Anwendung individuell anzupassen und zu erweitern. Durch die einfache Benutzeroberfläche ist diese intuitiv bedienbar.

6.2 Schwächen

Die Integration neuer Software in ein bestehendes System kann technische Herausforderungen mit sich bringen. Zum Beispiel muss sichergestellt werden, dass die bei STIHL vorhandenen Maschinenampeln (inkl. der OPC-Datenaufbereitung) kompatibel mit der entwickelten Lösung sind.

Durch die Ähnlichkeit der Anwendung zu bestehender Software bei STIHL können Mitarbeiterschulungen zwar reduziert, jedoch nicht ganz vermieden werden.

Aufgrund der begrenzten Entwicklungszeit und geringer Programmiererfahrung der Teammitglieder sind mit hoher Wahrscheinlichkeit nicht alle Fehler- und Ausnahmefälle behandelt. Weiter kann die Projektgruppe keine Aussagen bezüglich der Zuverlässigkeit, Sicherheit oder Speichereffizienz der Lösung treffen. Es muss davon ausgegangen werden, dass eine umfangreiche Testphase notwendig ist und diese entsprechend Ressourcen verbraucht. Mit einem kommerziellen System (vgl. Konzept A: WeAssist, aus Abgabe [1]) hätte STIHL diese Risiken an den Hersteller WERMA „verkauft“.

6.3 Chancen

Durch die Echtzeit-Aktualisierung wird eine frühzeitige Identifikation von Störungen in der Produktion ermöglicht. Weil Ressourcenausfälle durch rechtzeitiges Handeln gering gehalten werden, kann die Anwendung die Gesamtanlageneffektivität steigern. Durch die Reduzierung der Ausfallzeiten kann, im Sinne der Nachhaltigkeit, die Energieeffizienz der Produktion erhöht werden.

Auf Grundlage der gesammelten Maschinendaten können Visualisierungen zu den Prozessabläufen und einzelnen Maschinen erstellt werden. Dies kann zukünftige Entscheidungsfindungen durch Daten belegen und ggf. vereinfachen.

6.4 Risiken

Die Einführung einer neuen Software birgt Risiken. Ein Beispiel sind Akzeptanzprobleme bei den Mitarbeitenden. Diese können auftreten, wenn die Software nicht ausgereift ist und die Mitarbeitenden nicht zuverlässig damit arbeiten können.

Da das Programm nur in einer lokalen, virtuellen Umgebung getestet wird, können Folgekosten bei der Implementierung entstehen. Zur Integration in die Produktionsumgebung bei STIHL muss die Software angepasst werden. Probleme dabei können im Vorhinein nicht ausgeschlossen werden.

6.5 Handlungsempfehlungen für die Implementierung bei STIHL

Bevor die Software implementiert wird, sollte die Kompatibilität mit den bestehenden Strukturen und IT-Systemen von STIHL schrittweise getestet und überprüft werden. Dazu kann eine von der Produktion getrennte Testumgebung sinnvoll sein.

Am Anfang ist ein Pilotprojekt mit der Software empfehlenswert, um alle Funktionen zu testen. Zunächst sollte nur eine Maschine in die neue Software integriert werden. Im Falle einer fehlerhaften Funktion werden somit nur minimale Kapazitäten gebunden und Ressourcenausfälle minimiert.

Nach einem erfolgreichen Pilotprojekt kann die Anwendung weiter ausgerollt und die NodeRed-Implementierung stückweise ersetzt werden. Der Mehrwert der neuen Software wächst dann kontinuierlich. Durch ein schrittweises Vorgehen können Akzeptanzprobleme der Mitarbeitenden zudem reduziert werden.

7. Kostendarstellung

In Bezug auf Kapitel 5 aus der ersten Abgabe folgt die ausführliche Kostendarstellung [1, S. 12]. Diese gliedert sich in einmalige und laufende Kosten.

7.1 Einmalige Kosten

Da die benötigte Hardware (Mini-PCs, Server, Maschinenampeln und Visualisierungseinheiten) bei STIHL bereits vorhanden und implementiert ist, fallen dafür keine Kosten an. Die Maschinenampeln werden über die bestehende OPC-UA-Schnittstelle integriert. Neue Signalleuchten oder Steuereinheiten müssen demnach nicht neu angeschafft werden.

Das Projekt Visualisierung von Maschinenampeln ist eine Zusammenarbeit von STIHL und einer Studierendengruppe aus dem Master-Modul Digitalisierung an der TH Köln. Die Entwicklung der Software erfolgt durch die Studierenden. Diese verursachen keine Entwicklungskosten. Die Implementierung der Software bei STIHL wird von einem Projektingenieur für digitales Shopfloor Management übernommen. Somit entstehen keine direkten Implementierungskosten. Die Software basiert ausschließlich auf Open-Source Frameworks und Paketen. Lizenzkosten fallen daher nicht an. Für jede Produktionshalle muss am Anfang ein entsprechendes Dashboard mit den dort vorhandenen Maschinen und Maschinenampeln angelegt werden. Die damit verbundene Arbeitszeit wird vom zuständigen Projektingenieur bereitgestellt.

7.2 Laufende Kosten

Die Software benötigt regelmäßige Wartung, um Sicherheitslücken zu schließen und die Funktionalität zu erhalten. Dieser Prozess kann von dem Betreuer der Software von STIHL übernommen werden.

Die Mitarbeiter, die die Dashboards verwalten und aktualisieren, benötigen eine Schulung im Umgang mit der neuen Software. Da die Anwendung intuitiv gestaltet ist und sich an bestehenden Visualisierungstools orientiert, ist der Schulungsaufwand gering. Die Verwaltung und Erstellung neuer Dashboards können in den bestehenden Arbeitsablauf integriert werden. Ein verstärkter Personalbedarf ist nicht notwendig. Es entstehen daher keine zusätzlichen Kosten.

Wie in [1, S. 6 f.] angenommen, sind die von der Anwendung genutzten Server bereits im Einsatz und verursachen keine zusätzlichen Kosten.

7.3 Amortisation

Alle vorhandenen Aufgaben, die während und nach der Integration notwendig sind, werden von STIHL übernommen und in den alltäglichen Arbeitsablauf eingebunden [6]. Somit entstehen nur geringe Kosten für das Projekt „Visualisierung von Maschinenampeln“. Zusätzlich unterstützt die neue Software die nachhaltige Produktion durch den optimierten Einsatz von Ressourcen. Die frühzeitige Erkennung von Maschinenstörungen führt zu einer Reduzierung von Stillstandszeiten und einer besseren

Auslastung der Produktionsanlagen. Ausfälle sind lediglich während der Integration und Testphase zu erwarten. Die Software amortisiert sich in kürzester Zeit.

8. Fazit und Ausblick

Ziel des Projekts ist es, eine flexible Softwarelösung zu entwickeln, die eine einfache Benutzerführung und eine nahtlose Integration in die bestehende Produktionsumgebung bei STIHL ermöglicht. Die entwickelte Softwarelösung erfüllt dies vollumfänglich. Die Anwendung stellt in mehreren Punkten einen Mehrwert zur aktuellen Implementierung mit NodeRed dar. Das einfache Design fördert die intuitive Bedienbarkeit. Der verwendete Software-Stack baut auf dem Standard von STIHL auf und ermöglicht zudem eine sehr gute Skalierbarkeit. Zudem ermöglicht die offene Architektur einen flexiblen Einsatz sowie eine einfache Erweiterung der Anwendung.

Aufgrund der verfügbaren Ressourcen während der Entwicklung gibt es Optimierungspotenzial bei der Zuverlässigkeit, Sicherheit und der Speichereffizienz der Lösung. Die optionale Erweiterung (OPC-UA Tag Searcher, vgl. [1, S. 9]) wird wegen der begrenzten Zeit nicht mehr umgesetzt.

Aus den Handlungsempfehlungen lässt sich ableiten, dass die Chancen die Risiken überwiegen und ein Pilotprojekt weitere potenzielle Schwächen aufdecken kann. Die Einbindung von Feedback aller Akteure bietet eine gute Möglichkeit, die Software gezielt anzupassen, weiterzuentwickeln und gefundene Probleme zu beheben. Nach erfolgreicher Integration am Standort Weinsheim, ist der nächste übergeordnete Schritt die Skalierung der Lösung auf weitere STIHL-Standorte. Dies ermöglicht eine Standardisierung der Produktionsüberwachung. Die gesammelten Maschinendaten bieten eine Grundlage, um langfristig Muster und Zusammenhänge zu erkennen, ungeplante Stillstände zu reduzieren und die Effizienz der Produktion zu steigern.

Literaturverzeichnis

[1] [redacted] Steinker, [redacted] Chelly, [redacted] Zaafouri, [redacted] Feil, [redacted] Buhl, "Visualisierung von Maschinenampeln bei STIHL - Abgabe 1: Modul: Digitalisierung WiSe 24/25," TH Köln, Abgegeben am 16.11.2024.

[2] [redacted] Torsten [redacted], "Projektvorstellung: Entwicklung eines Tools zur Erstellung von Dashboards zur Visualisierung von Maschinenampeln", Persönliches Gespräch am 16. Oktober 2024.

[3] Torsten [redacted], "Besprechungsprotokoll 04", Regelmeeting, Okt. 2024.

[4] D. A. Norman und R. Verganti, "Incremental and Radical Innovation: Design Research vs. Technology and Meaning Change," *Design Issues*, 2014, doi: 10.1162/DESI_a_00250.

[5] M. Focke und J. Steinbeck, *Steigerung der Anlagenproduktivität durch OEE-Management: Definitionen, Vorgehen und Methoden,* 2. Aufl. (essentials). Wiesbaden: Springer Fachmedien Wiesbaden; Imprint Springer Gabler, 2024.

[6] Torsten [redacted], "Besprechungsprotokoll 13", Regelmeeting, Dez. 2024.

[7] Angular/Google (o.D.). "Angular V17 Dokumentation." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://v17.angular.io/docs

[8] OpenJS Foundation (o.D.). "node.js Website." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://nodejs.org/en

[9] Microsoft Corporation (o.D.). "ASP.NET 8 Dokumentation." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://learn.microsoft.com/de-de/aspnet/core/?view=aspnetcore-8.0

[10] Angular/Google (o.D.). "Angular components overview." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://v17.angular.io/guide/component-overview

[11] Torsten [redacted], "Besprechungsprotokoll 11", Regelmeeting, Nov. 2024.

[12] Nils Mehlhorn. "Angular Environments einrichten & testen." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://angular.de/artikel/angular-environments/

[13] Bohnhardt CAD-Consulting (o. D.). "Erstellung von Fabriklayouts." [Online.] Verfügbar: https://www.bohnhardt.net/autocadzeichenservice/fabriklayouts/

[14] Item Industrietechnik GmbH. "Signaltechnik: Die Sprache der Farben in der modernen Fertigung." Zugriff am: 4. November 2024. [Online.] Verfügbar: https://blog.item24.com/intralogistik/signaltechnik-die-sprache-der-farben-in-der-modernen-fertigung/

[15] Microsoft Corporation (o.D.). "NuGet Paket: Microsoft.EntityFrameworkCore.SqlServer." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.sqlserver/

[16] Microsoft Corporation (o.D.). "NuGet Paket: Microsoft.EntityFrameworkCore.Tools." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Tools

[17] Microsoft Corporation (o.D.). "NuGet Paket: OPCFoundation.NetStandard.Opc.Ua.Client." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://www.nuget.org/packages/OPCFoundation.NetStandard.Opc.Ua.Client/

[18] Microsoft Corporation (o.D.). "NuGet Paket: SkiaSharp." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://www.nuget.org/packages/SkiaSharp/

[19] Microsoft Corporation (o.D.). "NuGet Paket: Microsoft.AspNetCore.Mvc.NewtonsoftJson." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.NewtonsoftJson

[20] Microsoft Corporation. "Asynchrone Programmierszenarios." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://learn.microsoft.com/de-de/dotnet/csharp/asynchronous-programming/async-scenarios

[21] Rick Anderson. "Statische Dateien in ASP.NET Core." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://learn.microsoft.com/de-de/aspnet/core/fundamentals/static-files?view=aspnetcore-8.0

[22] Rick Anderson, Kirk Larkin, et. al. "Erstellen einer Web-API mit ASP.NET Core." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://learn.microsoft.com/de-de/aspnet/core/tutorials/first-web-api?view=aspnetcore-8.0&tabs=visual-studio

[23] Microsoft Corporation (o.D.). "Database design basics." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://support.microsoft.com/en-us/office/database-design-basics-eb2159cf-1e30-401a-8084-bd4f9c9ca1f5

[24] Google (o.D.). "Cloud SQL – Preise." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://cloud.google.com/sql/pricing?gad_source=1&gclid=CjwKCAiA0rW6BhAcEiwAQH28ItktO2n1GayMAdfTPMAtNjTl1--w0a17CbuQ-9cjchIs9vpy9QzumhoCdxMQAvD_BwE&gclsrc=aw.ds&hl=de

[25] Torsten [redacted], "Besprechungsprotokoll 14", Regelmeeting, Jan. 2025.

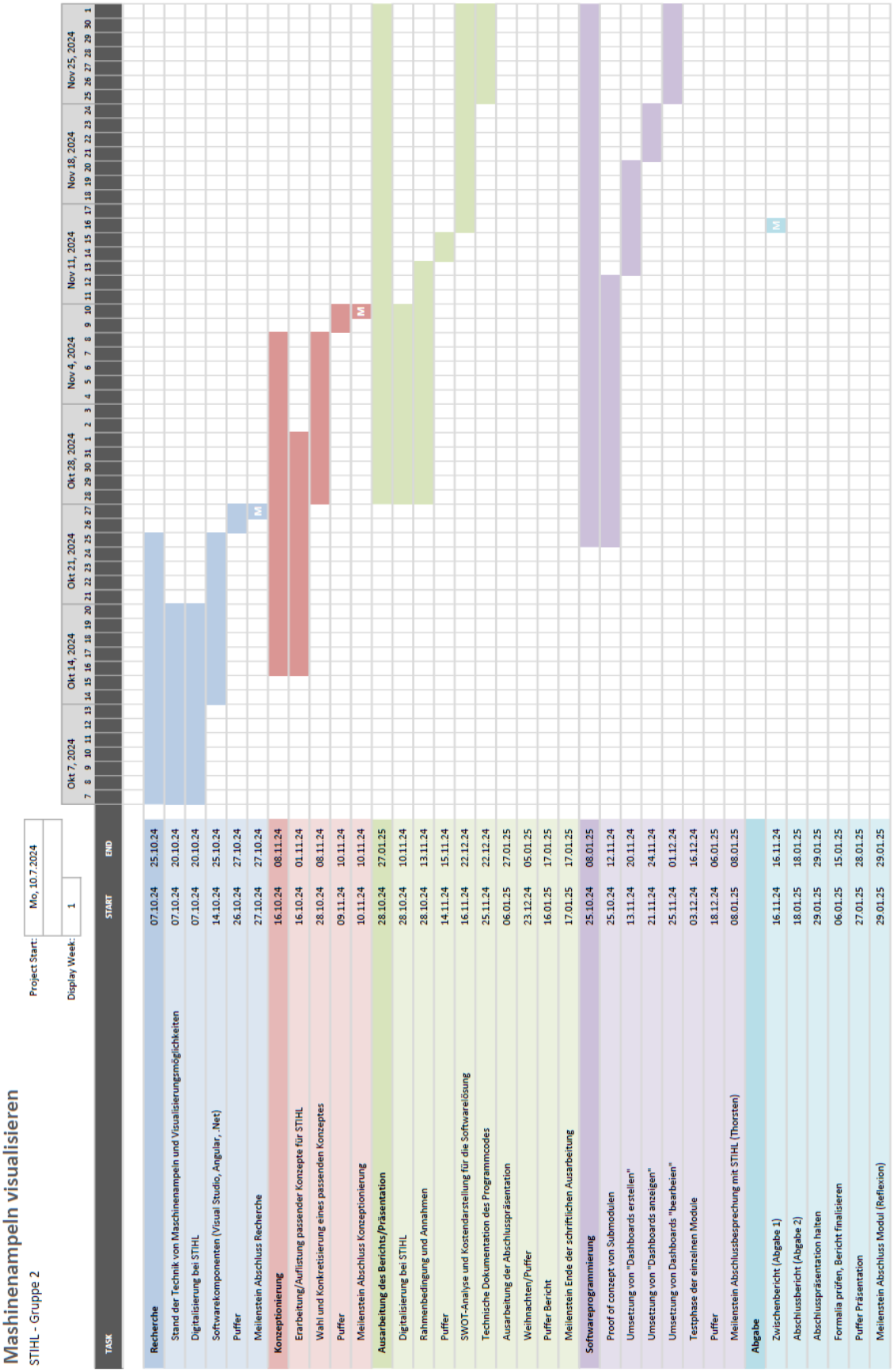
[26] Microsoft Corporation. "Dev cert export no longer creates folder." Zugriff am: 19. Dezember 2024. [Online.] Verfügbar: https://learn.microsoft.com/en-us/dotnet/core/compatibility/aspnet-core/9.0/certificate-export

[27] Bundesministerium des Innern und für Heimat. "SWOT Analyse." Zugriff am: 15. Januar 2025. [Online.] Verfügbar: https://www.orghandbuch.de/Webs/OHB/DE/OrganisationshandbuchNEU/4_MethodenUndTechniken/Methoden_A_bis_Z/SWOT_Analyse/swot_analyse_node.html

[28] Verein Deutscher Ingenieure e.V. "Digitalisierung als Enabler für Ressourceneffizienz." Zugriff am: 16. Oktober 2024. [Online.] Verfügbar: https://www.ressource-deutschland.de/themen/digitalisierung/

Anhang

Anhang 1: Gantt-Diagramm zur Projektplanung



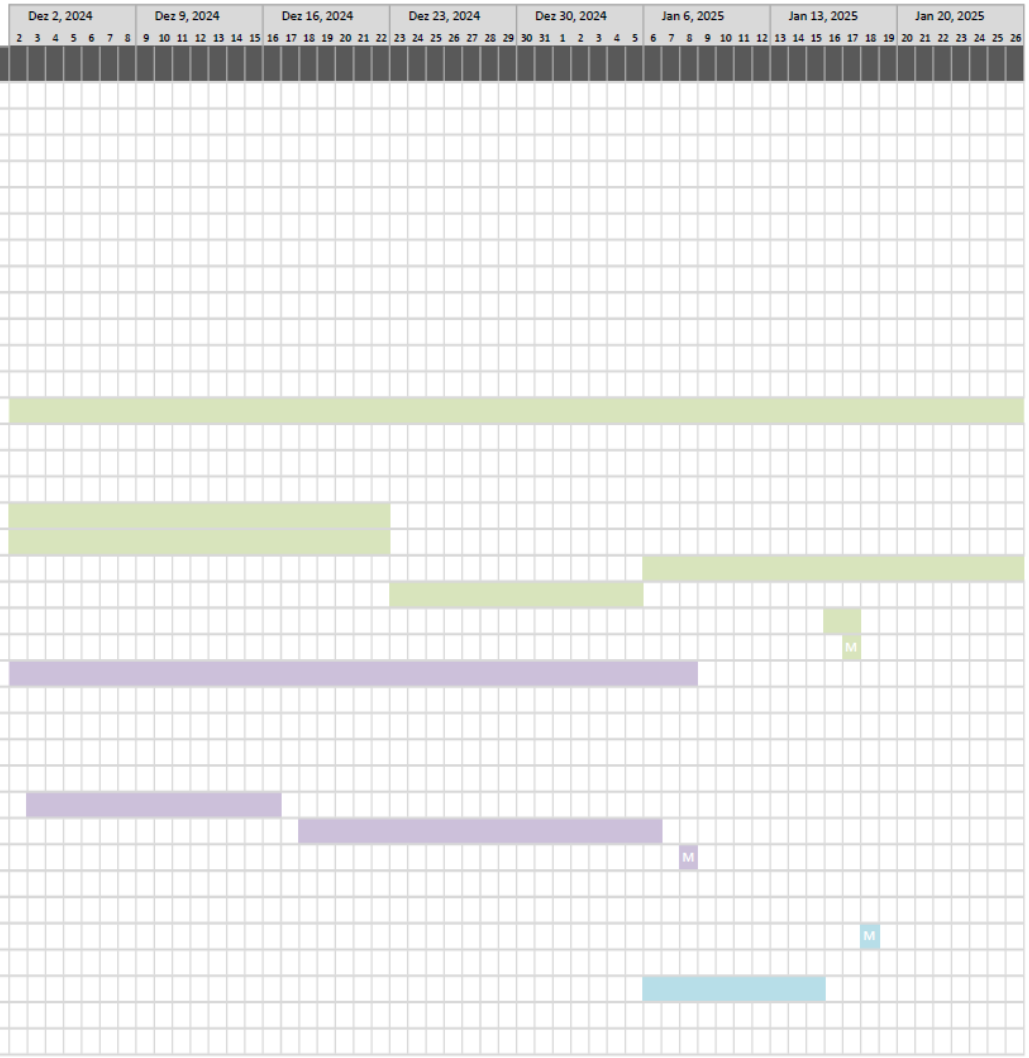
Mashinenampeln visualisieren

Seite 2/3

Project Start: Mo, 10.7.2024

Display Week: 9

TASK	START	END
Recherche	07.10.24	25.10.24
Stand der Technik von Maschinenampeln und Visualisierungsmöglichkeiten	07.10.24	20.10.24
Digitalisierung bei STIHL	07.10.24	20.10.24
Softwarekomponenten (Visual Studio, Angular, .Net)	14.10.24	25.10.24
Puffer	26.10.24	27.10.24
Meilenstein Abschluss Recherche	27.10.24	27.10.24
Konzeptionierung	16.10.24	08.11.24
Erarbeitung/Auflistung passender Konzepte für STIHL	16.10.24	01.11.24
Wahl und Konkretisierung eines passenden Konzeptes	28.10.24	08.11.24
Puffer	09.11.24	10.11.24
Meilenstein Abschluss Konzeptionierung	10.11.24	10.11.24
Ausarbeitung des Berichts/Präsentation	28.10.24	27.01.25
Digitalisierung bei STIHL	28.10.24	10.11.24
Rahmenbedingung und Annahmen	28.10.24	13.11.24
Puffer	14.11.24	15.11.24
SWOT-Analyse und Kostendarstellung für die Softwarelösung	16.11.24	22.12.24
Technische Dokumentation des Programmcodes	25.11.24	22.12.24
Ausarbeitung der Abschlusspräsentation	06.01.25	27.01.25
Weihnachten/Puffer	23.12.24	05.01.25
Puffer Bericht	16.01.25	17.01.25
Meilenstein Ende der schriftlichen Ausarbeitung	17.01.25	17.01.25
Softwareprogrammierung	25.10.24	08.01.25
Proof of conzept von Submodulen	25.10.24	12.11.24
Umsetzung von "Dashboards erstellen"	13.11.24	20.11.24
Umsetzung von "Dashboards anzeigen"	21.11.24	24.11.24
Umsetzung von Dashboards "bearbeien"	25.11.24	01.12.24
Testphase der einzelnen Module	03.12.24	16.12.24
Puffer	18.12.24	06.01.25
Meilenstein Abschlussbesprechung mit STIHL (Thorsten)	08.01.25	08.01.25
Abgabe		
Zwischenbericht (Abgabe 1)	16.11.24	16.11.24
Abschlussbericht (Abgabe 2)	18.01.25	18.01.25
Abschlusspräsentation halten	29.01.25	29.01.25
Formalia prüfen, Bericht finalisieren	06.01.25	15.01.25
Puffer Präsentation	27.01.25	28.01.25
Meilenstein Abschluss Modul (Reflexion)	29.01.25	29.01.25



Mashinenampeln visualisieren

Seite 3/3

Project Start: Mo, 10.7.2024

Display Week: 17

TASK	START	END
Recherche	07.10.24	25.10.24
Stand der Technik von Maschinenampeln und Visualisierungsmöglichkeiten	07.10.24	20.10.24
Digitalisierung bei STIHL	07.10.24	20.10.24
Softwarekomponenten (Visual Studio, Angular, .Net)	14.10.24	25.10.24
Puffer	26.10.24	27.10.24
Meilenstein Abschluss Recherche	27.10.24	27.10.24
Konzeptionierung	16.10.24	08.11.24
Erarbeitung/Auflistung passender Konzepte für STIHL	16.10.24	01.11.24
Wahl und Konkretisierung eines passenden Konzeptes	28.10.24	08.11.24
Puffer	09.11.24	10.11.24
Meilenstein Abschluss Konzeptionierung	10.11.24	10.11.24
Ausarbeitung des Berichts/Präsentation	28.10.24	27.01.25
Digitalisierung bei STIHL	28.10.24	10.11.24
Rahmenbedingung und Annahmen	28.10.24	13.11.24
Puffer	14.11.24	15.11.24
SWOT-Analyse und Kostendarstellung für die Softwarelösung	16.11.24	22.12.24
Technische Dokumentation des Programmcodes	25.11.24	22.12.24
Ausarbeitung der Abschlusspräsentation	06.01.25	27.01.25
Weihnachten/Puffer	23.12.24	05.01.25
Puffer Bericht	16.01.25	17.01.25
Meilenstein Ende der schriftlichen Ausarbeitung	17.01.25	17.01.25
Softwareprogrammierung	25.10.24	08.01.25
Proof of conzept von Submodulen	25.10.24	12.11.24
Umsetzung von "Dashboards erstellen"	13.11.24	20.11.24
Umsetzung von "Dashboards anzeigen"	21.11.24	24.11.24
Umsetzung von Dashboards "bearbeien"	25.11.24	01.12.24
Testphase der einzelnen Module	03.12.24	16.12.24
Puffer	18.12.24	06.01.25
Meilenstein Abschlussbesprechung mit STIHL (Thorsten)	08.01.25	08.01.25
Abgabe		
Zwischenbericht (Abgabe 1)	16.11.24	16.11.24
Abschlussbericht (Abgabe 2)	18.01.25	18.01.25
Abschlusspräsentation halten	29.01.25	29.01.25
Formalia prüfen, Bericht finalisieren	06.01.25	15.01.25
Puffer Präsentation	27.01.25	28.01.25
Meilenstein Abschluss Modul (Reflexion)	29.01.25	29.01.25

Anhang 2: Endpunkte und SQL-Operationen des Datenbankcontrollers

Endpunkt	SQL-Query	Input-Parameter	Antwort
api/DBController/getDashboards	`SELECT * FROM DashboardDB`	keine	Alle Einträge von DashboardDB
api/DBController/getAmpeln	`SELECT * FROM AmpelDB WHERE DASHBOARD_ID = @selected_ID`	ID des ausgewählten Dashboards	Einträge von AmpelDB mit gültiger Bedingung
api/DBController/addDashboard	`INSERT INTO DashboardDB (Name, IMG_PATH, aspectRatio) VALUES(@Name, @IMG_PATH, @aspectRatio)`	Name, Speicherpfad des Hallenlayouts, Seitenverhältnis des Hallenlayouts	OK/Fehler
api/DBController/addAmpel	`INSERT INTO AmpelDB (DASHBOARD_ID, POS_X, POS_Y, SIZE, ColorCount, COLORS, OPC_Addr, OPC_TagList) VALUES(@DASHBOARD_ID, @POS_X, @POS_Y, @SIZE, @ColorCount, @COLORS, @OPC_Addr, @OPC_TagList)`	Dashboard-ID, x_Position, y-Positon, Größe, Farbanzahl, Farbarray, OPC-Adresse, OPC-Tagarray	OK/Fehler
api/DBController/updateDashboardName	`UPDATE DashboardDB SET Name = @NewName WHERE ID = @ID`	Neuer Dashboardname, Dashboard-ID	OK/Fehler
api/DBController/updateAmpel	`UPDATE AmpelDB SET POS_X = @POS_X, POS_Y = @POS_Y, SIZE = @SIZE, ColorCount = @ColorCount, COLORS = @COLORS, OPC_Addr = @OPC_Addr, OPC_TagList = @OPC_TagList WHERE ID = @Dashboard_ID`	Dashboard-ID, x_Position, y-Positon, Größe, Farbanzahl, Farbarray, OPC-Adresse, OPC-Tagarray	OK/Fehler
api/DBController/deleteDashboard	`DELETE FROM DashboardDB WHERE ID = @ID`	Dashboard-ID	OK/Fehler
api/DBController/deleteAllAmpeln	`DELETE FROM AmpelDB WHERE DASHBOARD_ID = @selected_ID`	jeweilige Dashboard-ID; wird von *deleteDashboard()* aufgerufen	OK/Fehler
api/DBController/deleteAmpel	`DELETE FROM AmpelDB WHERE ID = @ID`	Ampel-ID	OK/Fehler

Anhang 3: SWOT-Analyse

Stärken	Schwächen
- Verbesserung der Transparenz und Abrufbarkeit vom Maschinenstatus - Förderung einer effizienten Kommunikation - Potenzielle Reduzierung von Stillstandszeiten - Anpassungsfähigkeit/Skalierbarkeit - Kompatibilität (nah an bisheriger Lösung) - Nutzen von vorhandener Infrastruktur - Geringe Kosten - Aufbau der Anwendung ist weltweit gleich und kann überall implementiert werden (Synergien zwischen Standorten) - Keine Abhängigkeit von Drittanbietern (offene Software) - Datenschutz - Benutzerfreundliche Oberfläche	- Integration im bestehenden System - Schulung von Mitarbeitern - Begrenzte Entwicklungszeit und Erfahrung - Nicht alle Fehler- und Ausnahmefälle behandelt/abgedeckt

Chancen	Risiken
- Potenzial zur Effizienzsteigerung durch frühzeitige Identifikation von Störungen - Vermeidung von Energieverbrauch durch rechtzeitige Problemerkennung (Nachhaltigkeit) - Bessere Entscheidungsfindung durch datengesteuerte Visualisierungen	- Akzeptanzprobleme bei Mitarbeitern - Folgekosten (Programm muss während der Implementierung ggf. Angepasst werden) - Keine Testphase in Produktionsumgebung

Anhang 4: Sitzungsprotokolle