Inhaltsverzeichnis

Inhaltsverzeichnis 1

1. Detaillierte Ausarbeitung und technische Dokumentation 2

1.1 Architekturübersicht 2

1.2 Frontend (Client) 4

1.2.1 Dashboard anzeigen 5

1.2.2 Dashboard updaten 9

1.2.3 Dashboard hinzufügen 10

1.3 Backend (Server) 11

1.3.1 Datenbank-Controller 13

1.3.2 Bild-Upload-Controller 14

1.3.3 OPC-Controller 14

1.4 SQL-Datenbank 16

2. Integration und Anwendung bei STIHL 17

2.1 Integration 17

2.2 Troubleshooting 19

Anhang X Endpunkte und SQL-Operationen des Datenbankcontrollers 20

3. Literatur 21

1. Detaillierte Ausarbeitung und technische Dokumentation

Ziel des Kapitels ist es, die Implementierung und technischen Details des Projekts transparent und nachvollziehbar zu dokumentieren.

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 eigebettete 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. [2]

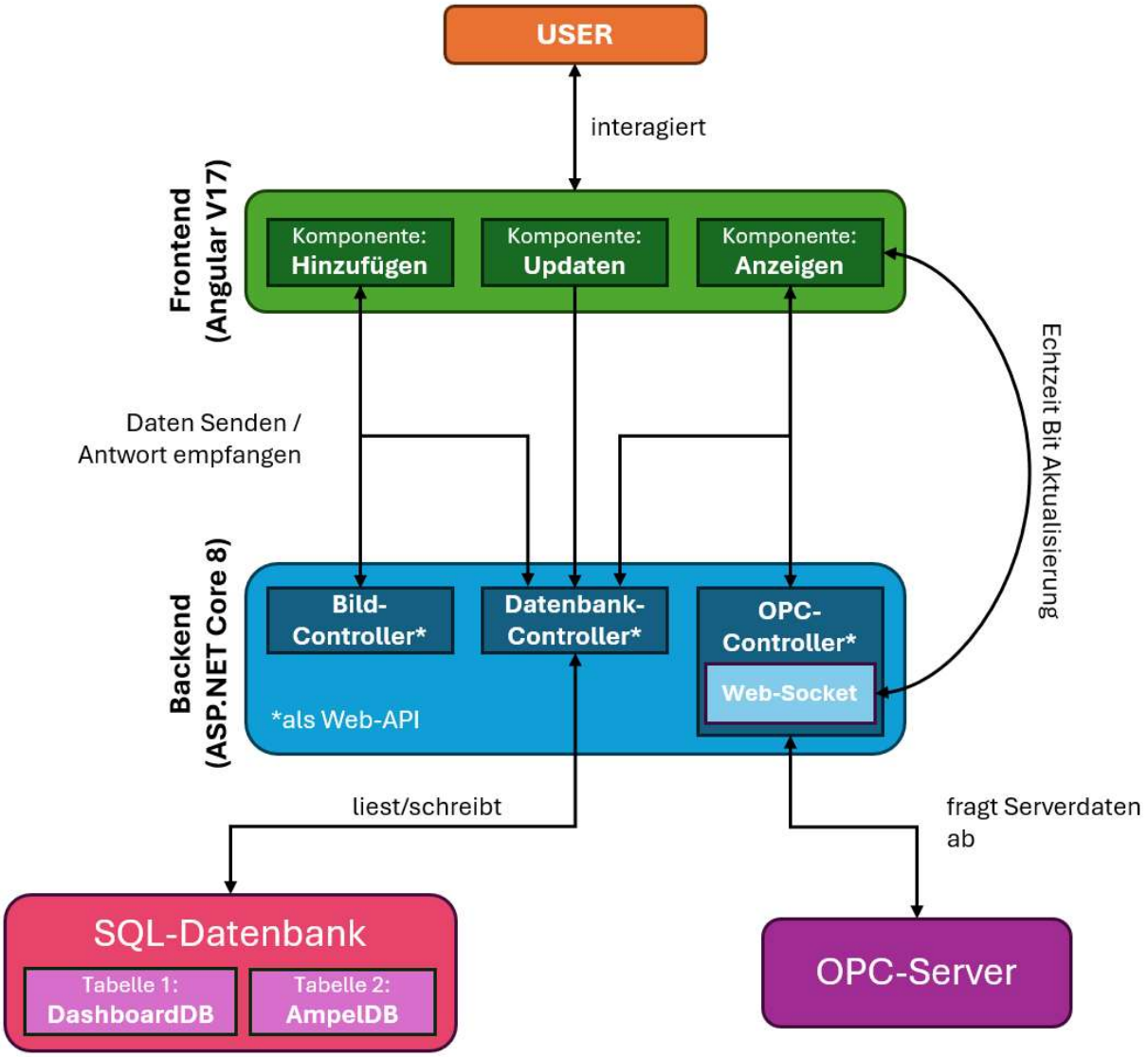
Der vollständige Code ist im GitHub-Repository *fynnbuhl/Maschinenampel* (siehe: https://github.com/fynnbuhl/Maschinenampel/) verfügbar.

In der Tabelle [verwendete Frameworks] sind die verwendeten Frameworks und deren Versionen aufgeführt. Diese Auswahl wurde von STIHL in der Projektbeschreibung vorgegeben. [3]

Frameworks	**Version**	**Beschreibung**
Angular CLI	17.3.10	Plattform zur Entwicklung clientseitiger Single-Page-Anwendungen [4]
Node.js	18.20.4	JavaScript Laufzeitumgebung [5]
ASP.NET Core	.NET 8.0	Als Backend (API) fungierendes Web-Framework [6]

1.1 Architekturübersicht

Die Abbildung [Architekturübersicht] veranschaulicht die Interaktion der wesentlichen Architekturbestandteile.



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 Bericht 1 genannten Anforderungsprofils (vgl. Kapitel 4.1 in [7]). Die einzelnen Funktionen werden in Kapitel 1.2 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 ein Web-Socket bereit, damit die Bits zum Visualisieren der Ampelzustände kontinuierlich dem Frontend bereitgestellt werden können.

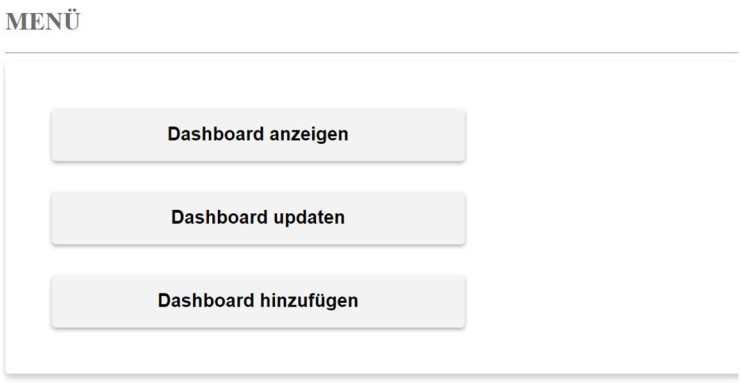
Die Architektur dieses Projekts wurde so gestaltet, dass Frontend, Backend und Datenbank klar voneinander getrennt sind. Dies ermöglicht eine skalierbare und wartungsfreundliche Umsetzung. Außerdem wird die Integration in die bestehende Produktionsumgebung von STIHL erleichtert [4], [6].

1.2 Frontend (Client)

Das Frontend wurde 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 [Verzeichnisstruktur vom Frontend] die wichtigsten Bestandteile der Verzeichnisstruktur im Frontend.

```
Frontend/
└── src/
    ├── app/
    │   ├── display-dashboard/
    │   │   └── display-dashboard.component
    │   ├── update-dashboard/
    │   │   └── update-dashboard.component
    │   ├── add-dashboard/
    │   │   └── add-dashboard.component
    │   ├── app.component
    │   ├── API-URL.service.ts
    │   └── websocket.service.ts
    └── environments/
        ├── environments.ts
        └── environments.development.ts
```

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 [UI Hauptmenü]). Für jeden Menü-Button ist ein *routerLink*-Attribut hinterlegt, welches den Nutzer beim Klicken die jeweilige Sub-Komponente (**-dahboard.component*) einblendet [8]. Durch die Bedingung `<div *ngIf="router.url === '/'">` in der root-Komponente wird das Menü ausgeblendet, sobald ein Routing zu einer anderen Komponente ausgewählt wurde. 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 [9].



Der Service *API-URL.service* wird von allen Sub-Komponenten benötigt. Dieser stellt die Serveradressen der verschiedenen Web-API-Controller (vgl. Abbildung Architektur) 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 diesem auf, um kontinuierlich mit dem Backend zu Kommunizieren. Die ausgelagerte Definition erhöht die Codelesbarkeit und reduziert den Wartungsaufwand innerhalb der Komponente.

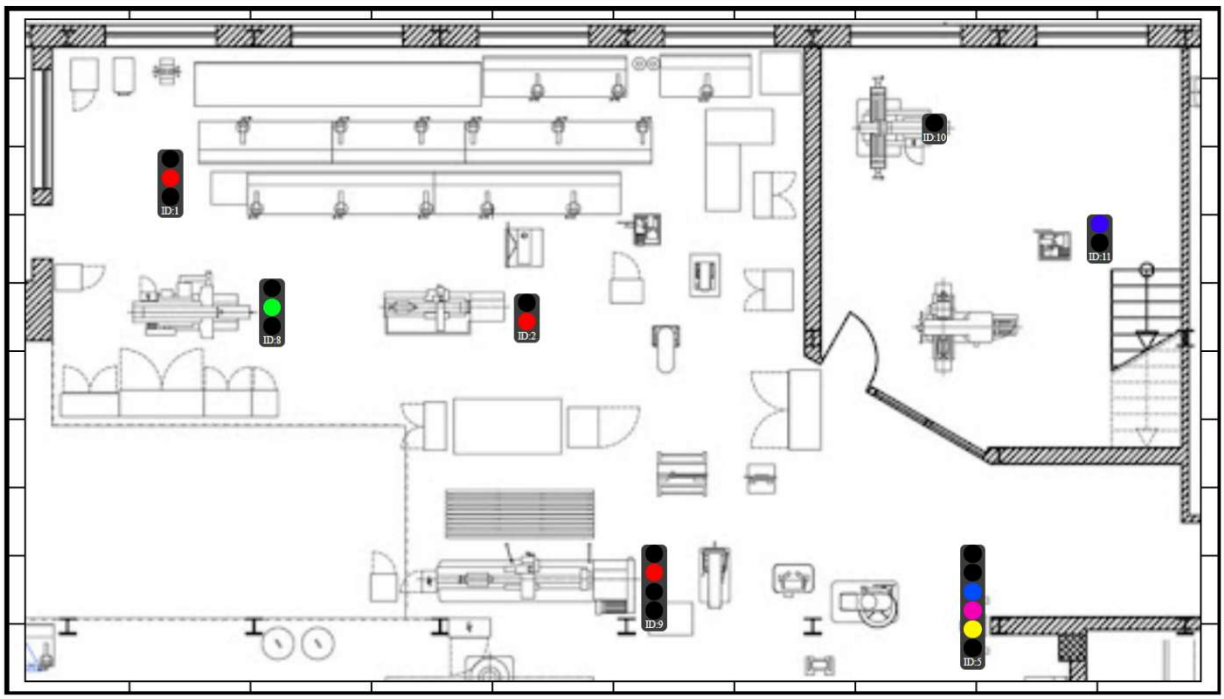
Der Ordner *Environments* enthält verschiedene Konfigurationen für unterschiedliche Umgebungen. Die Standartumgebung ist die Produktion (*environments.ts*). Zum Entwickeln wird die Umgebung Development (*environments.development.ts*) genutzt werden. Dies ermöglicht es, Umgebungsabhängigkeiten flexibel zu handhaben, ohne den Code anzupassen. Die entsprechende Umgebung kann durch ein Flag beim Erstellen festgelegt werden. [10]

1.2.1 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 [Funktionsweise des Menüs zur Dashboard Auswahl] dargestellten Blöcken.

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 [beispielhafte Visualisierung eines Dashboards] 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.



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 zeilenwiese über alle erhaltenden Ampeln. Zunächst wird der Antwort-String `COLORS` in ein zweidimensionales Array zerlegt und in `colorsArray: string[][]` gespeichert. Jede Zeile dieses Array 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 [Beispielhafter JSON-Body mit den Node-Adressen der abzufragenden Ampeln]). 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 [Beispielhafter JSON-Body der abgefragten Bit-Zustände]). 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.

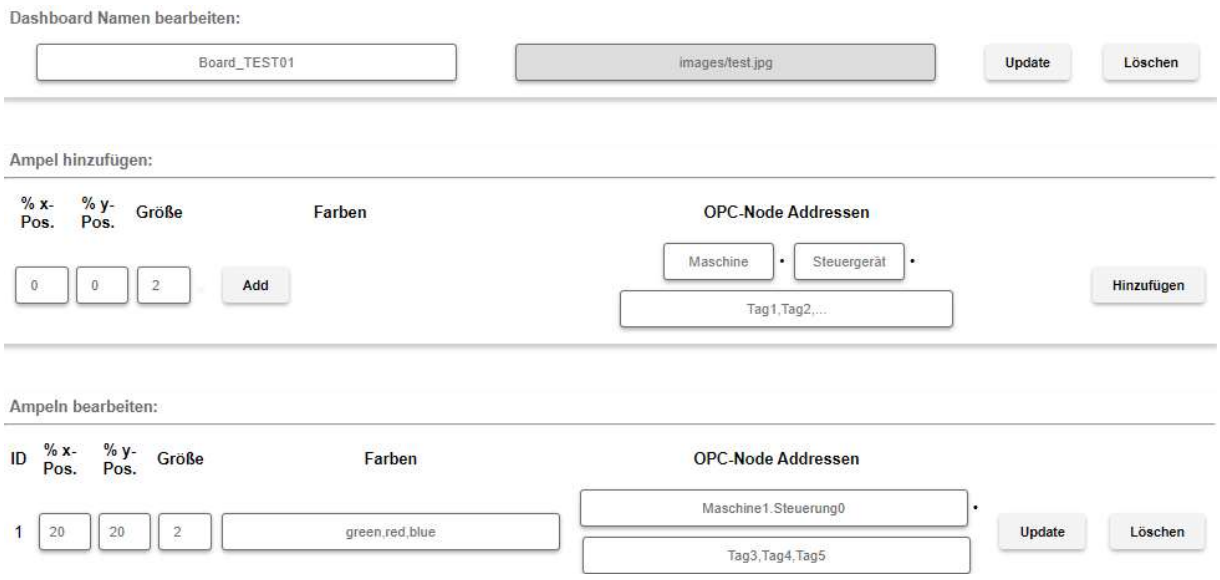
Das User-Interface visualisiert die Ampeln und ihre aktuellen Zustände in einem dynamisch generierten Layout. Wie in Abbildung [Codeausschnitt zur Visualisierung einzelner Ampeln] dargestellt, wird hierzu die **ngFor*-Direktive verwendet, wobei für jede Ampel eine HTML-Struktur der Klasse `ampel` erzeugt 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>

...
```

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.

1.2.2 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 [UI der Komponente Dashboard updaten] dargestellt eingeblendet.



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 werden 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 ersetz 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.

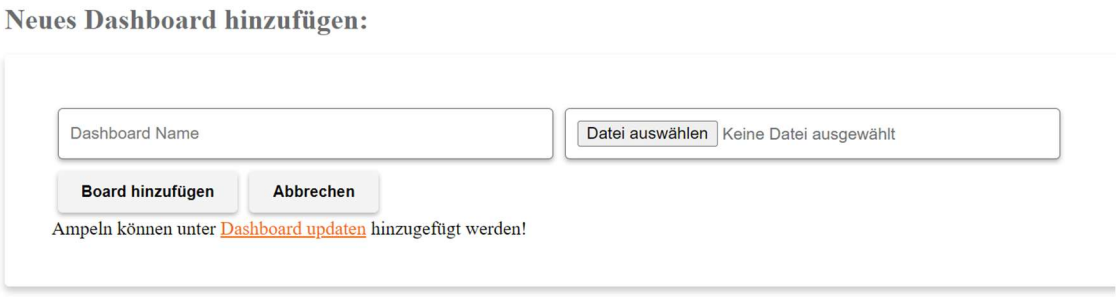
(B): Um eine neue Ampel dem Dashboard hinzuzufügen, muss die Eingabemaske ausgefüllt werden. Dabei können maximal sechs Farben ausgewählt werden (Kundenvorgabe [7]). Eine doppelte Auswahl ist nicht zulässig da dies der Norm DIN EN 60073 widerspricht [11]. Die OPC-Tags müssen mit Kommata voneinander getrennt 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 [beispielhafter JSON-Body einer API-Anfrage zum hinzufügen einer Ampel] 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"
}
```

(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 werden 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 jedoch die jeweilige Ampel-ID übergeben und die API-Endpunkte lauten *api/DBController/updateAmpel* und *api/DBController/deleteAmpel*.

1.2.3 Dashboard hinzufügen

Klickt der Nutzer im Menü auf „Dashboard hinzufügen" wird `routerLink="/addDashboard"` gesetzt. Das User-Interface dieser Komponente wird wie auf Abbildung [UI der Komponente Dashboard hinzufügen] dargestellt eingeblendet.



Hier kann der Anwender den neuen Dashboard-Namen eintragen und ein Bild (*.png, *.jpg, *.jpeg) des Hallenlayouts auswählen. Die Dateibeschränkungen wurden 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. Erst wenn der Speicherpfad und das Seitenverhältnis als Antwort vom Server erhalten wurde wird die Methode `saveToDB()` aufgerufen. Diese ruft ebenfalls per http POST-Request den API-Endpunkt *api/DBController/addDashboard* des Datenbank-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 wurden, wird die Komponente neu geladen um die Eingabefelder zurückzusetzten und der „Board hinzufügen“-Button wird wieder freigegeben. Der Nutzer wird über den erfolgten Vorgang informiert. Der „Abrechen“-Button lädt ebenfalls die Komponente neu um die Eingabefelder zurückzusetzten.

1.3 Backend (Server)

Das Backend des Projekts wurde 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 [Verzeichnisstruktur vom Backend] 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
```

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 war eine Anforderung von STIHL. [3]

Die zentralen Pakete der Anwendung sind:

- *Microsoft.EntityFrameworkCore.**SqlServer*:** ermöglicht die Anbindung an die SQL-Datenbank [12].
- *Microsoft.EntityFrameworkCore.**Tools*:** bietet Tools für die Datenbankmigration und -verwaltung [13].
- *OPCFoundation.NetStandard.**Opc.Ua.Client*:** wird für die Kommunikation mit dem OPC-UA-Server verwendet [14].
- ***SikaSharp*:** wird für die Verarbeitung und Speicherung von Bildmetadateien verwendet [15].
- *Microsoft.*AspNetCore.Mvc.**NewtonsoftJson:** wird für die Serialisierung und Deserialisierung von JSON-Daten verwendet [16].

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 wird in Kapitel 1.4 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.

1.3.1 Datenbank-Controller

Der Datenbank-Controller *DB_Controller*.cs dient als API-Schnittstelle für die Verwaltung und Abfrage von Daten der SQL-Datenbank. Er 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 [17]. Die Endpunkte der verwendeten CRUD-Operationen sind im Anhang [Endpunkte und SQL-Operationen des Datenbankcontrollers] aufgelistet.

1.3.2 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 [18]. 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 [Beispiel Antwort des Bild-Upload-Controllers]).

```
{
  "URL": "images/Hallenlayout01.png",
  "aspectRatio": 1.3333
}
```

1.3.3 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 [Beispielhaftes Adressarray mit den Node-Adressen der abzufragenden Ampeln] dargestellt.

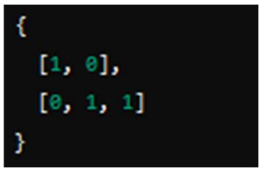
```
["Maschine1.Steuerung1.Tag1", "Maschine1.Steuerung1.Tag2"],
["Maschine2.Steuerung2.Tag1", "Maschine2.Steuerung2.Tag2", "Maschine2.Steuerung2.Tag3"]
```

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 zusammen mit der Verbindungs-URL an den Websocket zu senden wurde gegenüber einen dedizierten API-Endpunkts bevorzugt, da der Controller (inklusive der verwendeten Variablen) eine transiente Lebensdauer hat und bei jeder Anfrage eine neue Instanz erstellt wird. Daher wären die vom einen Endpunkt initialisieren Adressen nicht vom anderen Endpunkt verwendbar. Eine Variablen Deklaration als *static* würde verhindern, dass unterschiedliche Clients auf die Anwendung zugreifen. [19]

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 2). In Absprache mit STIHL wird eine *UserIdentity* und/oder ein *Sicherheitszertifikat* erst bei Bedarf, eigenständig im Service ergänzt [2].

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 [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.

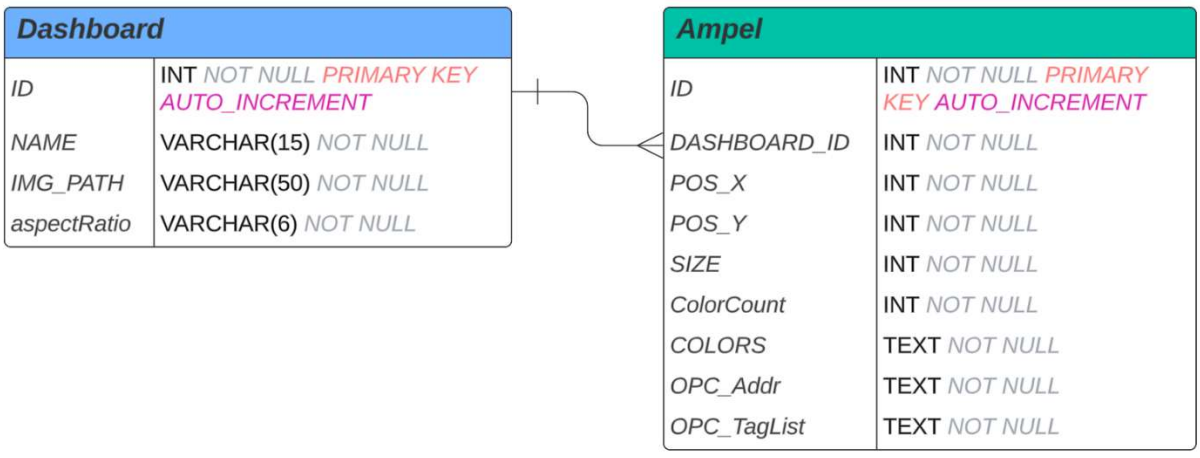
Das ausschließliche Übergeben der Ampel-Adressen des aktuellen Dashboards und das selektive Abfragen der zugehörigen OPC-Bits erhöht zwar leicht den Rechenaufwand der Controllerseite, sorgt aber dafür, dass der Client entlastet wird. Der Client müsste andernfalls die Bits aller auf dem OPC-Server vorhandener Nodes selektieren und mit den Ampeln verknüpfen. Die Rechenleistung des Backend-Servers ist jedoch höher ist als die der Clientseitigen Mini-PCs [20]. Um die vorhandenen Ressourcen effizient zu nutzen, wurde daher der oben beschriebene Ansatz gewählt.

1.4 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 [21]. Die verwendete Datenbank umfasst die zwei Tabellen:

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

Die Beziehungen der Entitäten *Dashboards* und *Ampeln* sind in Abbildung [Entity-Relationship-Modell der verwendeten SQL-Datenbank] dargestellt.



Für das vorliegende Projekt wird eine lokale SQL-Datenbank verwendet, es besteht auch die Möglichkeit, eine externen SQL-Server zu integrieren. Dazu muss die Verbindungszeichenfolge angepasst werden. Siehe dazu Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**. Der lokale Ansatz wurde gewählt, um die Komplexität so gering wie möglich zu halten. Dadurch kann das Testen vereinfacht und Cloud-Kosten gespart werden [22]

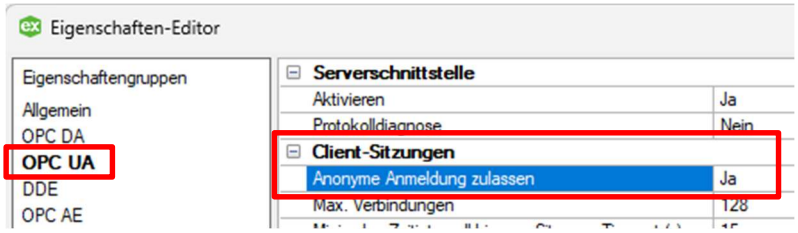
2. Integration und Anwendung bei STIHL

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 bei STIHL. Dieses Kapitel wurde nach Aufforderung von STIHL aufgenommen, um einen praxisnahen Leitfaden bereitzustellen. [23]
Es wird dabei vorausgesetzt, dass die in [7] Beschriebenen Rahmenbedingungen (Kapitel 3.2) und Voraussetzungen (Kapitel 3.3) durch STIHL erfüllt sind.

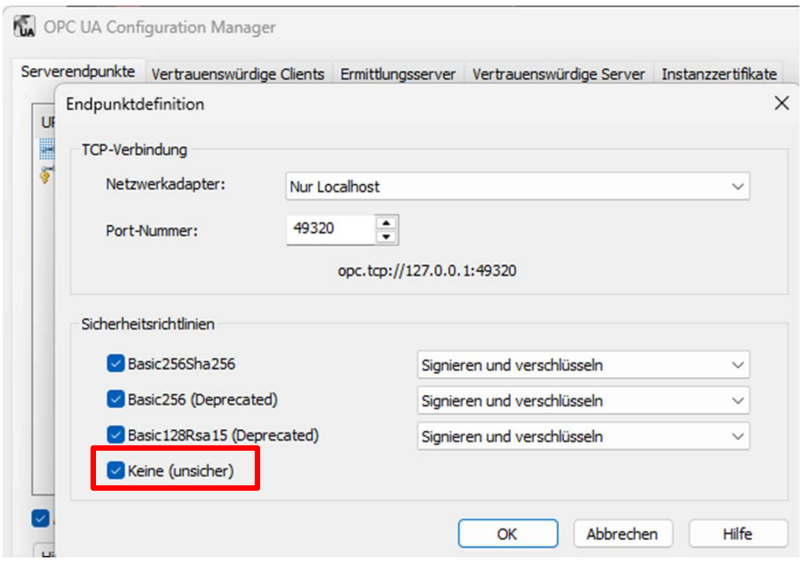
2.1 Integration

Zunächst 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 möglicherweise 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 KEPServer Eigenschaften-Editor)



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



Anschließend sind die verwendeten OPC-UA Serverdaten (URL, UpdateIntervall_ms, etc.) in den App-Settings zu aktualisieren (siehe: *Backend/appsettings.json*). Bei Verwendung einer externen SQL-Datenbank, muss diese verbreitet 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 müssen keine Einstellungen geändert werden. Zum Testen der Funktionalität ist das Front- und Backendserver zu starten und der Client im Browser aufzurufen. Währenddessen können die Debugging-Informationen der Konsolen überwacht werden.

2.2 Troubleshooting

Die Konsolen im Browser und auf dem Server zeigen Informationen über die laufenden Prozesse. 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 2.1).
- Durch die Statusmitteilung der Backend-Konsole die erfolgreiche Verbindung mit dem OPC-Server sicherstellen.
- Korrektheit des OPC_Bitarray überprüfen (vgl. Kapitel 1.3.3).

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 1.3.1 [verwendete Frameworks]).
    - Ab .NET Version 9 wird das Verzeichnis in welches das Zertifikat exportiert wird, nicht mehr erstellt, wenn dieses Verzeichnis nicht existiert [24].
- Das Verzeichnis *%APPDATA%\ASP.NET\https* prüfen und ggf. manuell erstellen.
- Anschießend alte Zertifikate löschen und neu generieren [24]:

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

Anhang X 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

3. Literatur

[1] STIHL AG & Co. KG. "Die Welt von STIHL, Über STIHL, Innovation." Zugriff am: 16. Oktober 2024. [Online.] Verfügbar: https://corporate.stihl.de/de/about-stihl/innovation

[2] Torsten Everts, "Besprechungsprotokoll 13", Regelmeeting, Dez. 2024.

[3] Evertz, Torsten Projektingenieur bei STIHL für digitales Shopfloor Management, "Projektvorstellung: Entwicklung eines Tools zur Erstellung von Dashboards zur Visualisierung von Maschinenampeln", Persönliches Gespräch am 16. Oktober 2024.

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

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

[6] 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

[7] Henning Steinker, Mahdi Chelly, Marwen Zaafouri, Kilian Feil, Fynn Buhl, "Visualisierung von Maschinenampeln bei STIHL - Abgabe 1: Modul: Digitalisierung WiSe 24/25," TH Köln, 2024.

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

[9] Torsten Everts, "Besprechungsprotokoll 11", Regelmeeting, Nov. 2024.

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

[11] 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/

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

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

[14] 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/

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

[16] 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

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

[18] 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

[19] 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

[20] Torsten Everts, "Besprechungsprotokoll 04", Regelmeeting, Okt. 2024.

[21] 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

[22] 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

[23] Torsten Everts, "Besprechungsprotokoll 14", Regelmeeting, Jan. 2025.

[24] 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