Inhaltsverzeichnis

1. Detaillierte Ausarbeitung des Konzeptes

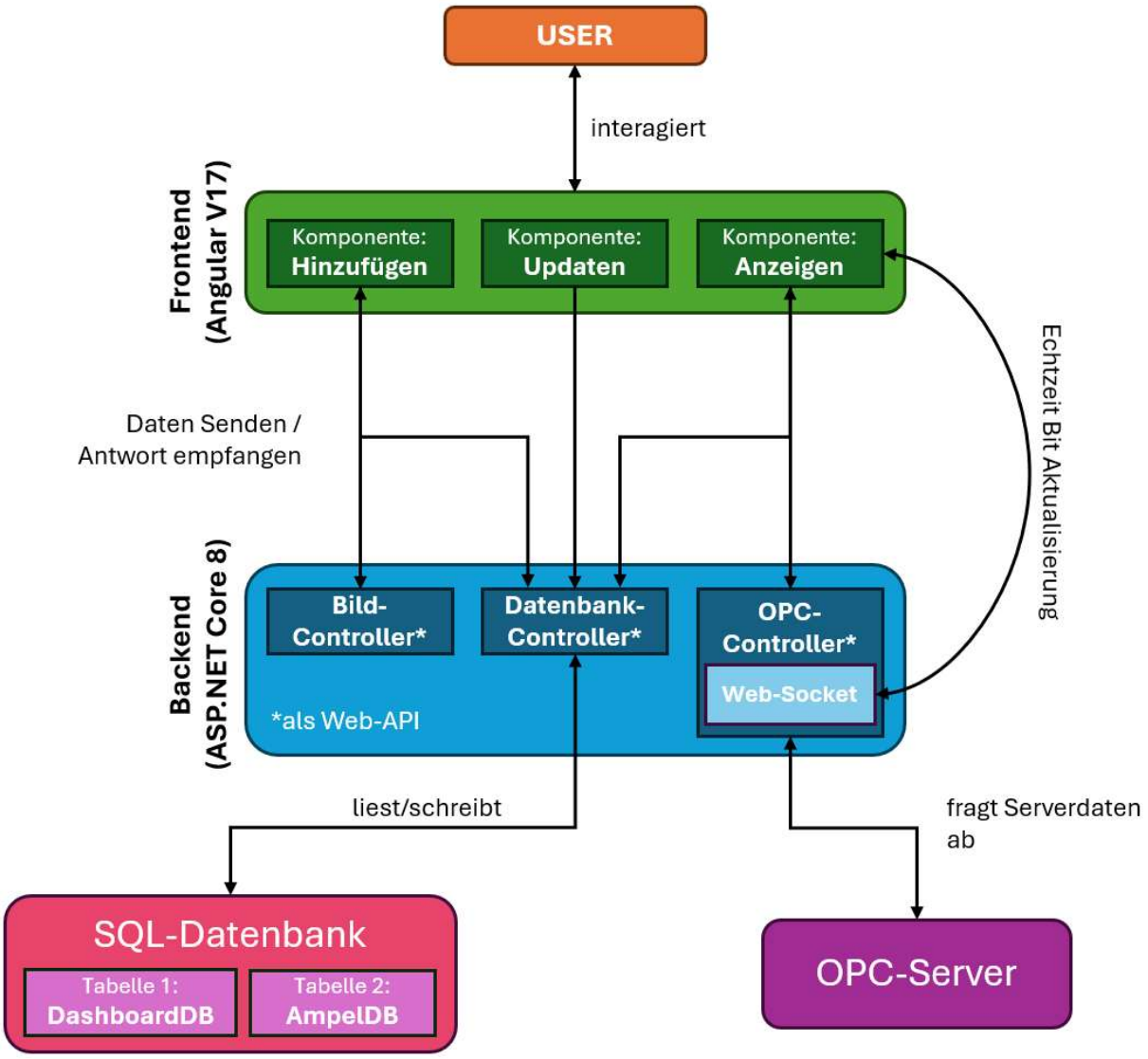
Dieses Kapitel beschreibt die Umsetzung der geplanten Softwarelösung. Das Projekt zielt darauf ab, eine flexible und benutzerfreundliche Lösung zu schaffen, die eine intuitive Erstellung und Anpassung von Dashboards ermöglicht. Auf jedem Dashboard sollen digitale Maschinenampeln die aktuellen Maschinenzustände der jeweiligen Produktionshalle in Echtzeit widerspiegeln. Ziel des Kapitels ist es, die Implementierung und technischen Details des Projekts transparent und nachvollziehbar zu machen.

Die Umsetzung basiert auf einem modernen Technologie-Stack, der sowohl im Frontend als auch im Backend effiziente und skalierbare Lösungen bietet. In der Tabelle [verwendete Frameworks] sind die verwendeten Frameworks und deren Versionen aufgeführt. Diese Auswahl wurde vom Kunden in der Projektbeschreibung vorgegeben.

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

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 Sub-Komponenten in Bezug auf die Dashboards zur Verfügung. Jede der drei Komponenten erfüllt einen Teilbereich des in Bericht 1 genannten Anforderungsprofils. Die einzelnen Funktionen werden in Kapitel 1.2 umfassend beschrieben. Allgemein gilt: Das Frontend sendet Daten des Nutzers an die jeweiligen Web-API-Controller des Backends. Die Controller verarbeiten diese und führen die ggf. notwendigen SQL- bzw.

OPC-Serverabfragen durch. 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, wartungsfreundliche Umsetzung und erleichtert die Integration in die bestehende Produktionsumgebung von STIHL.

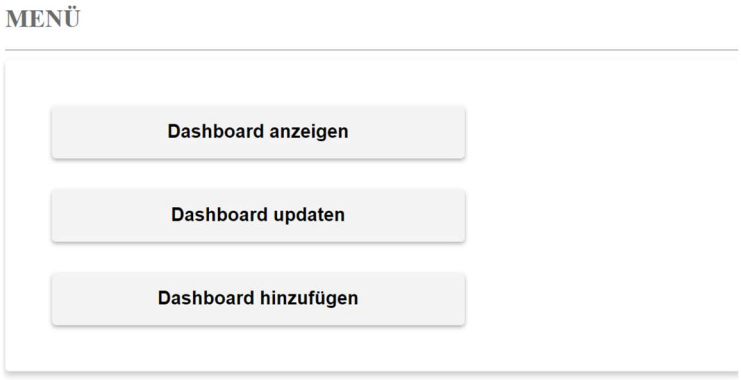
1.2 Frontend (Client)

Das Frontend wurde mit Angular entwickelt und stellt die Benutzeroberfläche zur Verfügung. Die Architektur des Clients folgt einer komponentenbasierten Struktur. Dies stellt eine klare Trennung von Logik und Darstellung sicher. In den folgenden Abschnitten wird detailliert 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 auch 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. Durch die Bedingung `<div *ngIf="router.url === '/'">` in der root-Komponente wird das Menü ausgeblendet, sobald ein Routing zu einer anderen Komponente ausgewählt wurde. Der Ansatz einzelne Kernfunktionen in jeweilige Sub-Komponenten auszulagern, wurde gewählt, um auch bei einer Funktionserweiterung eine konsistente Menüführung zu ermöglichen und die Anwendung an bestehende Tools bei STIHL anzupassen. Auf die Sub-Komponenten wird in separaten Kapiteln eingegangen (siehe Kapitel 1.2.1 - 1.2.3).



Der Service *API-URL.service* wird von allen Sub-Komponenten benötigt. Dieser stellt die jeweiligen 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.

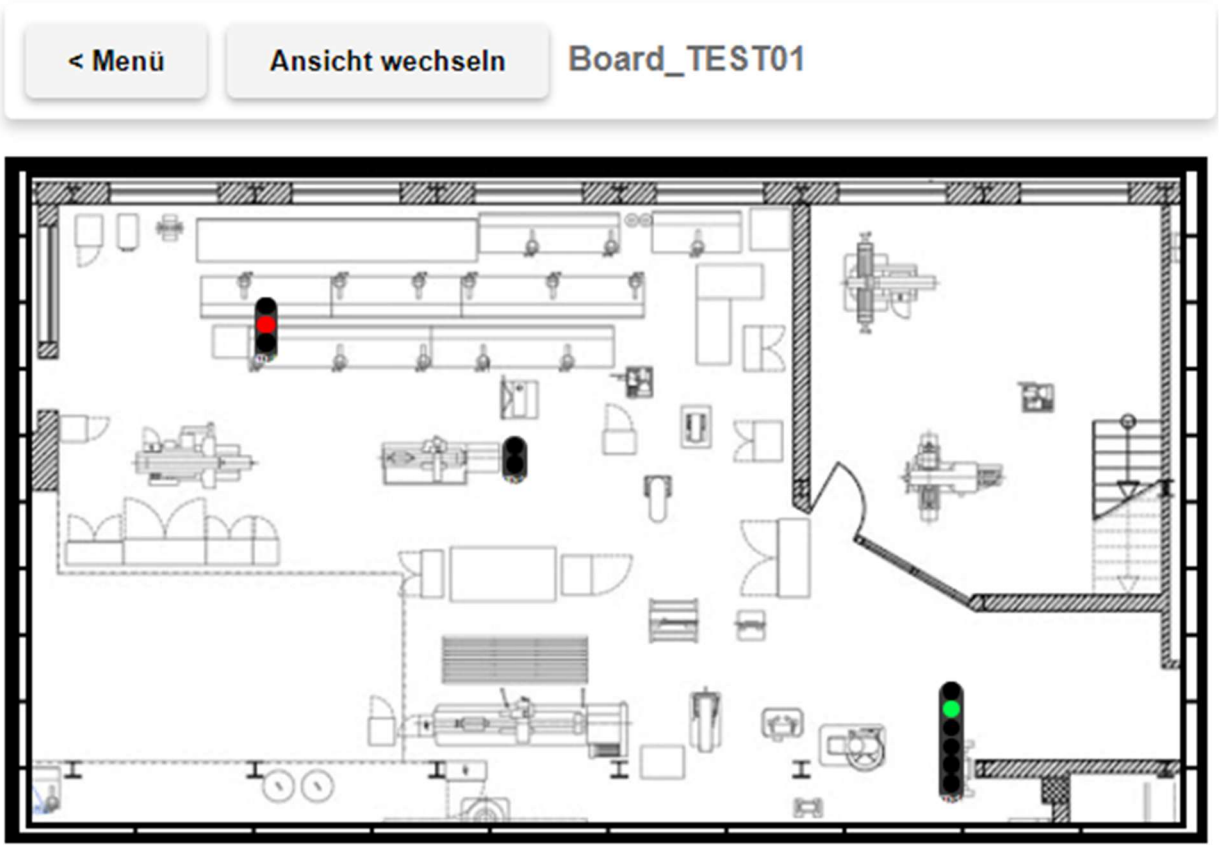
Der Service *websocket.service* definiert einen Web-Socket. Die Komponente *display-dashboard.component* ruft diesem zur Initialisierung des Web-Sockets zur kontinuierlichen Kommunikation mit dem Backend auf. Die ausgelagerte Definition erhöht die Codelesbarkeit und reduziert den Wartungsaufwand in der Komponente.

Der Ordner *Environments* enthält verschiedene Konfigurationen für unterschiedliche Umgebungen. Die Standartumgebung ist die Produktion (*environments.ts*), um entwickeln kann die Umgebung Development (*environments.development.ts*) genutzt werden. Dies ermöglicht es, Umgebungsabhängigkeiten flexibel zu handhaben, ohne den Code anpassen. Die Umgebung kann durch ein Flag beim Erstellen gesetzt werden.

1.2.1 Dashboard anzeigen

Klickt der Nutzer im Menü auf „Dashboard anzeigen“ wird `routerLink="/display-Dashboard"` gesetzt. Das User-Interface 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 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(ID: number, NAME: string, IMG_PATH: string, ratio: string)` 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] beispielhaft 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 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 Zeile. 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[][]`, welches den vollständigen Adressstring für alle OPC-Nodes der ange-

fragten Ampeln enthält (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 mit der Websocket-Verbindung an den OPC-Controller übergeben. Dieser Controller stellt dem Client die OPC-Bits in Echtzeit (Verzögerung von ~500ms sind möglich) bereit (vgl. Abbildung [Beispielhafter JSON-Body der abgefragten Bit-Zustände]). Die eingehenden Daten werden im `OPC_BITArray` gespeichert und do der aktuelle Ampelstatus visualisiert.

Der Hintergrund wird aus dem im Dashboard gespeicherten Bild-Pfad aus *wwwroot* vom Backend geladen. Nach dem Laden der aller Daten 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 optimal dargestellt wird.

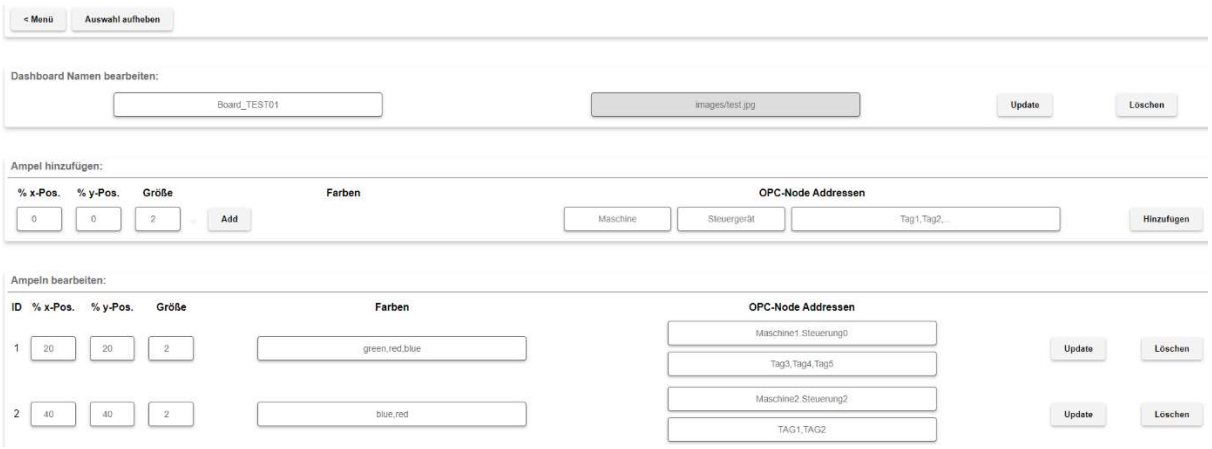
Das User-Interface visualisiert die Ampeln und ihre aktuellen Zustände in einem dynamisch generierten Layout. Wie in Abbildung [Codeausschnitt zur Visualisierung einzelner Ampeln] dargestellt, wird hierzu die **ngFor*-Direktive verwendet, wobei für jede Ampel eine HTML-Struktur der Klasse `ampel` erzeugt 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 runder 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 an 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 auf gleiche Art und Weise 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 die drei übergeordneten Aufgaben erledigt werden. Diese sind: (A) Dashboard Namen bearbeiten, (B) Ampel hinzufügen und (C) 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 neu-Positionierung 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 zunächst die Eingabemaske ausgefüllt werden. Dabei können maximal sechs Farben ausgewählt werden. Eine doppelte Auswahl ist nicht zulässig da dies der Norm DIN EN 60073 widerspricht. 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 wie in (A) die `selectedID` des ausgewählten Dashboards übergeben und ein http POST-Request an den API-Endpunkt *api/DBController/addAmpel* gesendet. Ein beispielhafter Body im JSON-Format ist in Abbildung [beispielhafter JSON-Body einer API-Anfrage zu hinzufügen einer Ampel] dargestellt.

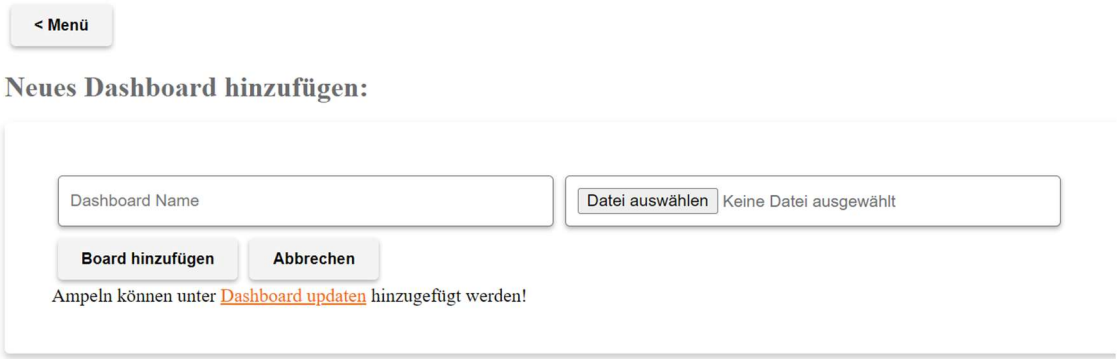
```json
{
  "Dashboard_ID": 1,
  "POS_X": 50,
  "POS_Y": 20,
  "SIZE": 3,
  "ColorCount": 3,
  "COLORS": "red,green,blue",
  "OPC_Addr": "Maschine01.Steuer01",
  "OPC_TagList": "Tag1,Tag2,Tag3"
}
```

(C): In diesem Abschnitt sind alle dem aktuellen Dashboard zugeordneten Ampeln mit ihren frei veränderbaren Eigenschaften aufgelistet. Über die Eingabemaske können diese Eigenschaften verändert werden. Die einzelnen Farben müssen dabei mit Kommata getrennt werden und müssen 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="/add-Dashboard"` 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 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 „Abrechen“-Button lädt ebenfalls die Komponente neu um die Eingabefelder manuell 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 und verarbeitet die eingehenden Anfragen. Im Folgenden werden die Controller sowie deren Interaktion mit der SQL-Datenbank und dem OPC-UA-Server detailliert beschrieben. Dabei visualisiert Abbildung [Verzeichnisstruktur vom Backend] die wichtigsten Bestandteile der Verzeichnisstruktur im Backend.

```
Backend/
├── Pakete/
│   ├── SQLServer
│   ├── Tools
│   ├── Opc.Ua.Client
│   ├── ImageSharp
│   └── ...
├── 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, diese sind essenziell für die Funktionalität der Anwendung. Alle Pakete sind Open-Source und *MIT*-Lizensiert. Dazu gehören:

- *Microsoft.EntityFrameworkCore.**SqlServer***: Ermöglicht die Anbindung an die SQL-Datenbank.

- *Microsoft.EntityFrameworkCore.**Tools***: Bietet Tools für die Datenbankmigration und -verwaltung.
- *OPCFoundation.NetStandard.**Opc.Ua.Client***: Wird für die Kommunikation mit dem OPC-UA-Server verwendet.
- *SixLabors.**ImageSharp***: Wird für die Verarbeitung und Speicherung von Bildmetadateien verwendet.
- *Microsoft.*AspNetCore.Mvc.**NewtonsoftJson:** Wird für die Serialisierung und Deserialisierung von JSON-Daten verwendet.

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 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 zwei 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. Die verwendeten CRUD-Operationen sind in der Tabelle [Endpunkte und SQL-Operationen des Datenbankcontrollers] aufgelistet.

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

1.3.2 Bild-Upload-Controller

Der Bild-Upload-Controller *imgUploadController.cs* speichert die hochgeladenen Bilddateien der Frontend-Komponente *add-dashboard.component*. Er stellt eine API bereit, die über den Endpunkt *api/imgUpload/upload* Bilddateien entgegennimmt. Jeder Datei wird im Verzeichnis *wwwroot/images* gespeichert. Dieses Verzeichnis ist das Stammverzeichnis für statische Inhalte und verbessert die Sicherheit und Wartbarkeit der Anwendung. Zur Vermeidung von Namenskonflikten wird der Dateiname durch einen eindeutigen Zeitstempel ergänzt. Zusätzlich werden mithilfe der Bibliothek ImageSharp 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 externen OPC-UA Server. 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 Frontend und Backend.

Gleichzeitig werden über die Verbindungs-URL auch die Adressstrings der zu visualisieren Ampeln 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 das zweidimensionale Bitarray `int[][] OPC_BitArray` mit dem Wert *-1* initialisiert. Durch diesen Wert können Fehler bei der OPC-Server-Abfrage schneller erkannt werden, da *-1* kein gültiger Bitzustand ist.

Dieser Ansatz wurde gegenüber eines 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.

Bei aktivem Web-Socket wird außerdem eine Verbindung zum OPC-UA-Server hergestellt. Dazu wird eine Instanz des *OPC_Service* initalisiert. Der Web-socket ruft alle 500 Millisekunden die Controller-Methode `UpdateBits(OPC_AddrArray, OPC_BitArray)` auf. Diese iteriert über alle Elemente des Adressarrays und fragt den jeweiligen Bitzustand (0/1) vom OPC-Server ab. Der jeweilige Wert wird schließlich im Bitarray aktualisiert. Sobald die neuen Daten zur Verfügung stehen, sendet der Web-Socket das Bitarray im JSON-Format zurück an den Client (vgl. Abbildung [Beispielhafter JSON-Body der abgefragten Bit-Zustände]).

```
{
  [1, 0],
  [0, 1, 1]
}
```

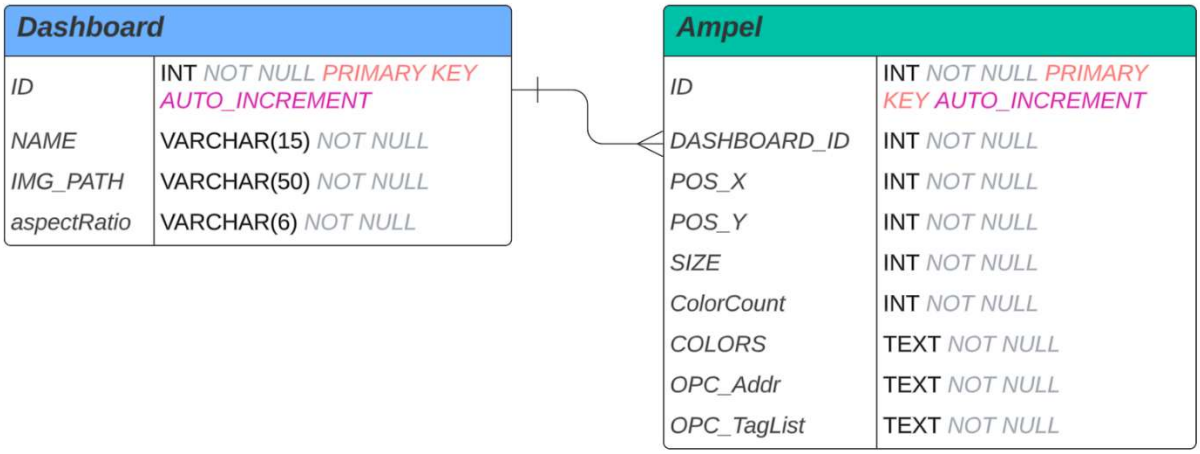
Das ausschließliche Übergeben der Adressen für das aktuelle Dashboard und das selektive Abfragen der zugehörigen OPC-Bits erhöht den zwar leicht den Rechenaufwand auf der Controllerseite, sorgt aber dafür, dass der Client entlastet wird. Der Client müsste andernfalls die Bits aller auf dem OPC-Server vorhandener Nodes selektieren und mit den Ampeln verknüpfen. Da davon auszugehen ist, dass die Rechenleistung des Backend-Servers höher ist als die der Clientseitigen Mini-PCs wurde 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. Die verwendete Datenbank umfasst de 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, Farbarray, OPC-Node-Adressen, OPC-Tagarray)

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



Für das vorliegende Projekt wird eine lokale SQL-Datenbank verwendet, es besteht jedoch auch die Möglichkeit, eine externen SQL-Server zu integrieren. 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.

2. Implementierung und Anwendung bei STIHL

- Visual Studio Projektordner in dediziertem Pfad speichern
- In allen Environments die Server-URL inklusive Port des Backend-Servers aktualisieren (siehe: `serverUrl: '<BackendServerURL>:<PORT>'`)
- Die Zugangsdaten des OPC-UA Servers in xyz aktualisieren
- Bei Verwendung einer externen SQL-Datenbank (also, wenn nicht die im Projekt enthaltene lokale Datenbank verwendet werden soll):
    - Die Tabellen *DashboardDB* und *AmpelDB* mit der jeweiligen SQL-Query (siehe: *Backend/Datenbank/SQLQuery_<Tabellenname>.sql*) erstellen
    - Verbindungszeichenfolge der zu verwendenden SQL-Datenbank in den App-Settings aktualisieren (siehe: *Backend/appsettings.json:* `"ConnectionStrings": {"DBConnect": "<Verbindungszeichenfolge>"}`)
- Zum Testen der Funktionalität: Anwendung starten und das Frontend im Browser aufrufen

2.1 erste Schritte

1. Dashboard anlegen
2. Ampeln hinzufügen
3. Dashboards anzeigen