Hochschule Esslingen
University of Applied Sciences

# Entwicklung eines elektronischen Logbuchs für den Test von Batteriezellen

**Bachelorarbeit**

im Studiengang
Softwaretechnik und Medieninformatik

vorgelegt von

**Max Manfred Meier**
Matr.-Nr.: 754468

am 28. Juli 2021
an der Hochschule Esslingen
in Kooperation mit der
SYSTECS Informationssysteme GmbH

| | |
|---|---|
| Erstprüfer: | Prof. Dr. rer. nat. Mirko Sonntag |
| Zweitprüfer: | Prof. Dr.-Ing. Kai Warendorf |

# Kurzfassung

Im Zuge der Verkehrswende werden in Zukunft viele neue Elektrofahrzeuge auf den Markt kommen. Diese Fahrzeuge enthalten Batterien, die zuvor auf ihre Eigenschaften getestet werden müssen. Deshalb werden bereits Testfabriken gebaut.

Gegenstand dieser Bachelorarbeit ist der Entwurf und die Implementierung des elektronischen Logbuchs, einer Teilanwendung der SYSTECS Test Factory.

Dieses Logbuch soll es erlauben, generische Event Typen zu erstellen, die anschließend geloggt werden können. Es soll dabei möglich sein, dass die Typen mehrere selbst definierte Attribute enthalten, um beispielsweise auch die Messwerte der Batteriezellen mitzuspeichern.

Die Nutzer der Test Factory sollen über eine Benutzeroberfläche auf das elektronische Logbuch zugreifen können. Ebenfalls sollen andere Teilapplikationen der Test Factory über eine REST-API darauf zugreifen.

# Abstract

In the course of the traffic turnaround, many new electric vehicles will come onto the market in the future. These vehicles contain batteries that must first be tested for their properties. That's why test factories are already being built.

The subject of this bachelor thesis is the design and implementation of the of the electronic logbook, a partial application of the SYSTECS Test Factory.

In this logbook, generic event types are to be created, which can then be logged. It should be possible for the types to contain several self-defined attributes, for example to also store measured values of the battery cells.

The users of the Test Factory should be able to access the electronic logbook via a web application and other applications of the Test Factory should be able to access the electronic logbook via a REST API.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis



# Codeverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| API | Application Programming Interface |
| CI/CD | Continuous Integration / Continuous Delivery |
| DevOps | Development Operations |
| DLL | Dynamic Linked Library |
| EF Core | Entity Framework Core |
| ELO | Elektronisches Logbuch |
| FK | Fremdschlüssel, Foreign Key |
| GUID | Globally Unique Identifier |
| LINQ | Language Integrated Query |
| OAS | OpenAPI Specification |
| PK | Primärschlüssel, Primary Key |
| PMC | Package Manager Console |
| REST | Representational State Transfer |
| SQL | Structured Query Language |
| UI | User Interface |

# 1 Einführung

In den nächsten Jahren wird die Automobilindustrie viele neue Elektrofahrzeuge auf den Markt bringen. Diese Fahrzeuge enthalten eine Batterie, die aus mehreren Batteriezellen bestehen. Bevor diese Batteriezellen ein Fahrzeug antreiben können, müssen sie umfangreich getestet werden. Ziel dieser Tests ist es, Eigenschaften der Batteriezellen zu untersuchen. Die Eigenschaften sind beispielsweise stark abhängig von der Temperatur und von der zyklischen & kalendarischen Alterung der Batteriezelle. In den Datenblättern der Batteriezellen sind diese Abhängigkeiten nicht oder nur grob angegeben. (vgl. Zurawka, et al., 2019)

Diese Eigenschaften sind unter anderem notwendig, um den aktuellen Ladungsinhalt der Batterie während der Fahrt zu bestimmen. Es ist zwar bekannt, welche Ladung zu jeder Zeit entnommen wird, aber der aktuelle Ladungsinhalt kann aufgrund dieser Eigenschaften nur grob berechnet werden. (vgl. Zurawka, et al., 2019)

Um die Batteriezellen auf ihre Eigenschaften zu testen, werden derzeit und in Zukunft Prüffabriken für Batteriezellen gebaut. Eine solche, bereits gebaute Prüffabrik eines deutschen Automobilherstellers besteht aus ca. 200 Prüfkammern (Klima- oder Temperaturschränke) mit jeweils 50 Prüfständen. Da an jedem Prüfstand eine Batteriezelle getestet werden kann, kann eine solche Prüffabrik ca. 10000 Batteriezellen gleichzeitig prüfen. (vgl. Zurawka, et al., 2019)

Die aktuellen Lösungen hierfür bestehen aus mehreren Anwendungen von verschiedenen Wettbewerbern, sodass die einzelnen Insellösungen nicht einfach vernetzbar sind. (vgl. Zurawka, et al., 2019)

Die SYSTECS Informationssysteme GmbH entwickelt hierfür eine innovative Prüfanlage, bestehend aus Hard- und Software zur durchgängigen, voll automatisierten, selbst lernenden Prüfung von Batteriezellen. Sie trägt den Namen SYSTECS Test Factory. (vgl. Zurawka, et al., 2019)

In dieser Prüfanlage sollen mehrere Batterien gleichzeitig vollautomatisch auf ihre Eigenschaften geprüft werden. Zur Übersicht ist es dazu notwendig, dass der Status der einzelnen zu testenden Batterien in einem einheitlichen System protokolliert wird.

Diese Arbeit beschäftigt sich mit der Entwicklung eines elektronischen Logbuchs für den Test dieser Batteriezellen.

## 1.1 Problemstellung

In der Prüffabrik werden mehrere Batterien, die aus verschiedenen Batteriezellen bestehen können, getestet. Hierfür wird ein System benötigt, dass es jeder Batterie erlaubt, ihren Status und die dazugehörigen Messwerte zu loggen. Gleichzeitig soll das System die Status anderer Anwendungen der Test Factory am gleichen Ort protokollieren. Dies erlaubt beispielsweise der Teilapplikation Battery Cell (UuT) neu erstellte Cell Types zu loggen (siehe Anhang B, Abbildung 15).

Es wird demzufolge ein generisches System benötigt, welches eigene Typen von Events definieren kann, um anschließend Einträge dieser definierten Event Typen zu loggen. Jeder Typ benötigt ebenfalls die Möglichkeit, eigene Attribute zu definieren, um beispielsweise Messwerte der Batteriezellen loggen zu können. Da diese Messwerte verschiedene Datentypen enthalten können, muss es möglich sein, dass die Attribute diese Datentypen unterstützen.

Das elektronische Logbuch soll gleichzeitig auch unterstützen, dass sowohl Anwender über eine Benutzeroberfläche als auch die anderen Anwendungen der Test Factory über die REST-API auf das Logbuch zugreifen können. Die Benutzeroberfläche soll dazu die gleiche REST-API benutzen.

## 1.2 Herausforderung

Die Herausforderung dieses Projektes besteht darin, ein geeignetes Datenmodell zu entwerfen, sodass das Datenmodell die benötigten Anforderungen erfüllt. Diese Aufgabe wird näher im Kapitel 4.2 erläutert.

Außerdem soll die REST-API so aufgebaut sein, dass sie von allen anderen Anwendungen der Test Factory verwendet werden kann. Die REST-API soll sowohl den Anwendern als auch den Applikationen der Test Factory erlauben auf das elektronische Logbuch zuzugreifen.

Des Weiteren soll die Benutzeroberfläche zum Design und der Usability der anderen Anwendungen der Test Factory passen. Für die Webanwendung soll das neue Web-Framework ASP.NET Core Blazor Server verwendet werden.

## 1.3 Aufbau dieser Arbeit

Diese Arbeit ist in sechs Kapitel unterteilt, wobei das erste Kapitel als Einleitung in diese Arbeit dient.

Das zweite Kapitel zeigt die Grundlagen dieser Arbeit auf. Hier werden die, für das Verständnis der Arbeit, wichtigen Technologien erläutert.

Das dritte Kapitel beschäftigt sich mit den funktionalen und nicht funktionalen Anforderungen an das elektronische Logbuch. Diese Anforderungen werden anhand von Use-Case-Diagrammen visualisiert.

Das vierte Kapitel beschreibt die Implementierung des elektronischen Logbuchs und geht dabei auf die Architektur des elektronischen Logbuchs ein. Anschließend werden das generische Datenmodell und die Entitäten beschrieben. Danach beschreibt das Kapitel den Aufbau und die Implementierung des Webservices und der Webanwendung. Am Ende des Abschnittes wird noch der Buildprozess und die Fehlerbehandlung der Anwendung erläutert.

Das fünfte Kapitel evaluiert die zuvor beschriebene Implementierung basierend auf ihren Anforderungen aus dem dritten Kapitel.

Das letzte Kapitel fasst diese Arbeit zusammen und gibt einen Ausblick auf die zukünftige Entwicklung des elektronischen Logbuchs und der Test Factory.

# 2 Grundlagen

Dieses Kapitel beschäftigt sich mit den Grundlagen dieser Arbeit und beschreibt die Software und die Technologien, die zum Einsatz kommen.

## 2.1 Azure DevOps

DevOps ist eine Komposition aus Development und Operations. Es beschreibt die Vereinigung von Prozessen, Technologien und Zusammenarbeit von Mitarbeitern aus Entwicklung und Betrieb, um hochwertige Produkte zu erschaffen. DevOps wirkt sich auf die Qualität der Software, kürzere Entwicklungszeiten und Testbarkeit der Anwendung aus. Der DevOps Anwendungslebenszyklus (siehe Abbildung 1) ist dabei in vier Phasen unterteilt: Planung, Entwicklung, Bereitstellung und Betrieb. (vgl. Microsoft Corporation, o.D.)

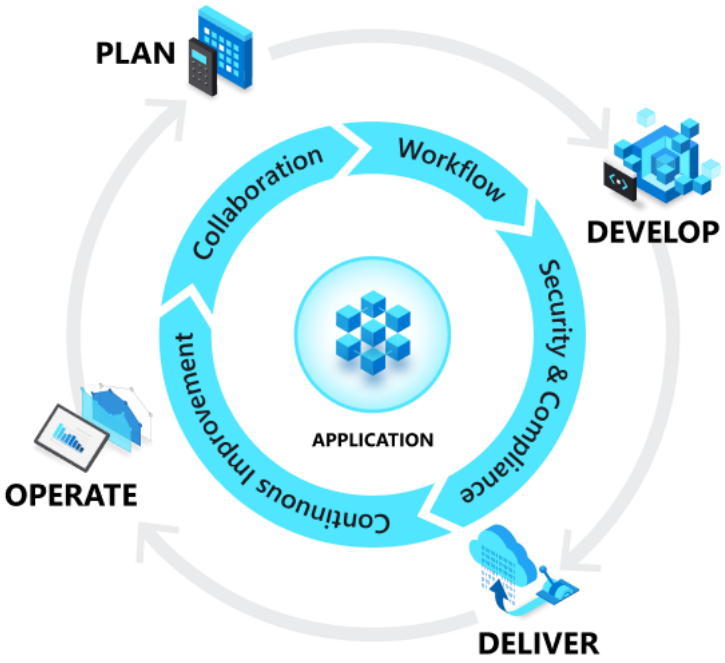


Abbildung 1: DevOps Anwendungslebenszyklus (Microsoft Corporation, o.D.)

Azure DevOps unterstützt die Entwickler bei der Teamarbeit, Planung und bei der Erstellung und Bereitstellung gemeinsamer Anwendungen. Azure DevOps bietet mehrere eigenständige Dienstleistungen an:

- **Azure Repos** stellen Git Repositories zur Verfügung.
- **Azure Pipelines** stellen Build- und Release-Dienste zur Verfügung, um Continuous Integration und Continuous Delivery (CI/CD) zu unterstützen.

- **Azure Boards** bestehen aus agilen Scrum- und Kanban-Tools zur Planung, Nachverfolgung von Code, Fehlern und Problemen.
- **Azure Test Plans** bieten Tools zum Testen der Applikationen.
- **Azure Artifacts** ermöglicht das Teilen von Paketen, wie zum Beispiel npm-Pakete oder NuGet-Pakete von privaten und öffentlichen Quellen freizugeben und in die Pipelines zu integrieren.

Außerdem bietet Azure DevOps die Möglichkeit, Informationen über Wiki-Seiten zu dokumentieren. (vgl. Urban, et al., 2021)

## 2.2 Azure

Microsofts Cloudplattform Azure ist derzeit nach Amazon AWS die zweitgrößte Cloudplattform und hatte im vierten Quartal 2020 einen Marktanteil von 20%. Azure bietet über 200 Produkten an, wie zum Beispiel Datenbanken, App Services, Azure Functions und mehr. Die verschiedenen Produkte können an verschiedenen Standorten gehostet werden, um bestimmte Märkte einfacher zu bedienen. (vgl. Synergy Research Group, 2021)

In dieser Arbeit wurde eine Azure SQL-Datenbank und mehrere App Services, zum Hosten von Web Applikationen und REST APIs, genutzt.

### 2.2.1 Azure SQL Database

Die Azure SQL-Datenbank ist eine relationale, skalierbare Datenbank, die für die Cloud gebaut und optimiert wurde. Die Datenbank basiert auf dem SQL Server und ist mit diesem quasi kompatibel. Gegenüber dem SQL Server hat die Azure Datenbank aber einige Vorteile: Der Entwickler spart sich Zeit beim Verwalten der Datenbank, da man nicht die Datenbank Engine, das Betriebssystem und die Hardware verwalten muss. Die Datenbank ist über das Azure Portal konfigurierbar. (vgl. Ovhal, et al., 2020)

### 2.2.2 Azure App Service

Der Azure App Service ist ein Service zum Hosten von Web Applikationen, REST APIs und mobile Backends. Der App Service unterstützt Anwendungen in verschiedensten Sprachen, wie .NET, .NET Core, Java, Ruby, Node.js, PHP oder Python in einer Windows- oder Linux-basierenden Umgebung. Zusammen mit Azure DevOps können die Azure App Services aktuell gehalten werden, da in der Azure Pipeline, beispielsweise nach einem erfolgreichem Pull Request, direkt deployed werden kann. Auch der Azure App Service ist über das Azure Portal konfigurierbar. (vgl. Lin, et al., 2020)

## 2.3 Entity Framework Core

Entitiy Framework Core (EF Core) ist ein Objekt-Datenbank-Mapper für .NET. Mit EF Core kann ein Entwickler mit .NET Objekten Queries über LINQ (Language Integrated Query) abfragen. Dies vereinfacht dabei den Datenbankzugriff. EF Core unterstützt verschiedene Datenbanken, wie SQL Server, Azure SQL Database, SQLite, Cosmos DB, MySQL und PostgreSQL. (vgl. Vickers, et al., 2020)

Das Framework unterstützt drei Ansätze wie .NET mit der Datenbank interagieren kann:

- Der **Code First** Ansatz wird verwendet, wenn der Entwickler eine neue Datenbank erstellen und dabei die Entitäten im Code beschreiben möchte. Hier erstellt der Entwickler die Objekte und den `DbContext` im Code. Mithilfe einer Code-First-Migration kann der Entwickler daraus eine Datenbank generieren. Updates können über eine neue Migration eingespielt werden.
  Bei diesem Ansatz kann sehr schnell eine neue Datenbank generiert werden. Er wird damit oft bei kleineren Anwendungen benutzt.
- Der **Model First** Ansatz wird benutzt, wenn man eine neue Datenbank mit der Oberfläche in Visual Studio erstellen möchte. Hier kann der Entwickler die Modelle mit dem Entity Framework Designer erstellen. Der Designer generiert anschließend den Code, über den danach identisch wie beim Code First Ansatz die Datenbank über eine Migration generiert werden kann.
- Der **Database First** Ansatz wird verwendet, wenn eine vorhandene Datenbank genutzt werden soll. Aus dieser Datenbank kann über die Konsole mit dem Befehl: „`dotnet ef dbcontext scaffold`“, die Entitätsobjekte und die `DBContext` Klasse generiert werden. Wenn die Datenbank geändert wurde, kann der Entwickler die Modelle neu von der Datenbank erstellen lassen. Die alten Modelle werden dann überschrieben.
  Dieser Ansatz hat den Vorteil, dass der Entwickler die Datenbank auf Performance abstimmen kann, deswegen wird dieser Ansatz oft bei größeren Projekten verwendet.

Diese Arbeit nutz den Database First Ansatz, um mehr Kontrolle über die Datenbank zu erhalten.

## 2.4 Automapper

Der Automapper ist ein Objekt-Mapper, der zum Mapping von Eingabeobjekten zu Ausgabeobjekten eingesetzt wird (vgl. Bargaoanu, et al., 2020). In dieser Arbeit wird der Automapper verwendet, um Datamodels in das jeweilige Domainmodel umzuwandeln und umgekehrt. Im folgenden Codeteil sieht man eine beispielhafte

Konfiguration des Automappers. Zur besseren Anschaulichkeit wurde nur das Mapping zwischen den Datenmodel und Domainmodel der Klasse `EventTypeAttribute` dargestellt. Man benötigt eine Klasse, die von der Basisklasse `Profile` erbt. Im Konstruktor dieser Klasse kann der Entwickler dann das Mapping konfigurieren. Alle Properties der Objekte, die den gleichen Namen und Datentyp haben, werden automatisch richtig gemappt. Wenn es Abweichungen gibt, kann man über die Methode `ForMember()` das Mapping anpassen.

```
public class MappingProfileElectronicLogBook : Profile
{
    public MappingProfileElectronicLogBook()
    {
        ...
        MapEventTypeAttribute();
        ...
    }

    private void MapEventTypeAttribute()
    {
        CreateMap<EventTypeAttributeDto, EloEventTypeAttribute>()
           .ForMember(dest => dest.DataType, opt => opt.Ignore());
        CreateMap<EloEventTypeAttribute, EventTypeAttributeDto>();
    }
}
```

Codeteil 1: Mapping Konfiguration

Die Konfiguration des Mappings wird am Start der Anwendung in der Methode `ConfigureServices()` aufgerufen.

## 2.5 Swashbuckle

Swashbuckle ist eine Implementierung der OpenAPI Spezifikation (OAS) für .NET (vgl. Morris, 2021). Bei der OAS handelt es sich um eine standardisierte, sprachunabhängige Spezifikation für das Beschreiben von RESTful APIs (vgl. SmartBear Software, 2020).

Mit Swashbuckle kann basierend auf der OAS eine automatisierte Dokumentation einer REST-API erstellt werden. Mit einem XML-Kommentar und Annotation-Attributen im Code kann die Dokumentation bearbeitet werden.

Aus dem folgenden Codeteil wird dann die Dokumentation generiert, die in Abbildung 2 und Abbildung 3 zu sehen ist. In diesem Beispiel wurden die Zusammenfassungen der Endpoints, die Beschreibung der Parameter und die möglichen Rückgabetypen des Endpoints `/api/electroniclogbook/datatypes/{id}` dokumentiert.

Swashbuckle generiert außerdem auch eine Dokumentation im JSON Format.

```
/// <summary>
///     Returns a DataType.
/// </summary>
/// <param name="id">The Id of the DataType</param>
/// <returns>DataType</returns>
/// <response code="200">Returns a DataType.</response>
/// <response code="204">
///      Returns null if DataType is not found.
/// </response>
[HttpGet(DataTypesWithIdPath)]
[Produces("application/json")]
[ProducesResponseType(typeof(DataType), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
public async Task<ActionResult<DataType>> GetDataTypeAsync(Guid id)
{
    var result = await
           _dataTypeService.GetDataTypeAsync(id).ConfigureAwait(false);

    return result;
}
```

Codeteil 2: Swashbuckle Beispiel

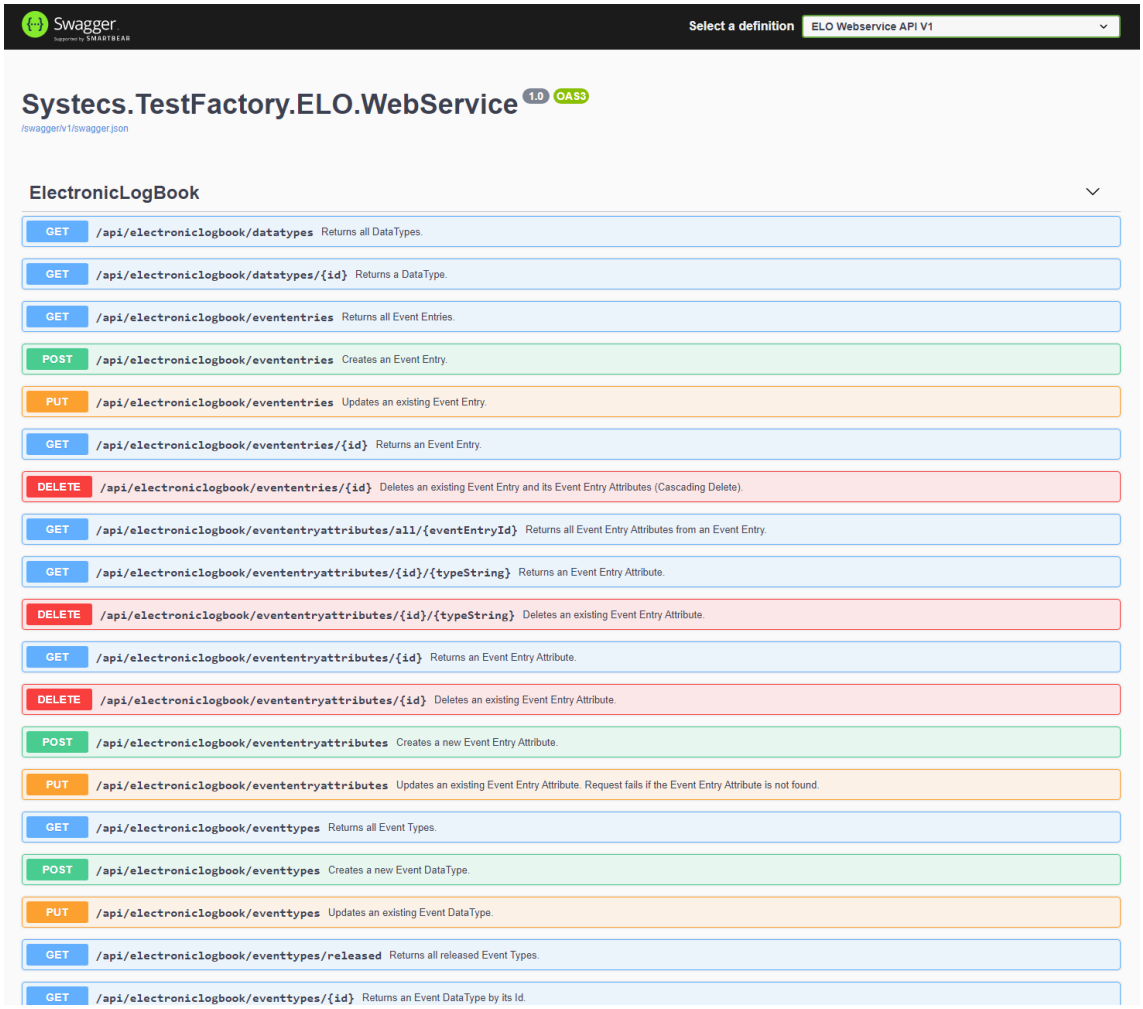


Abbildung 2: Swashbuckle Dokumentation

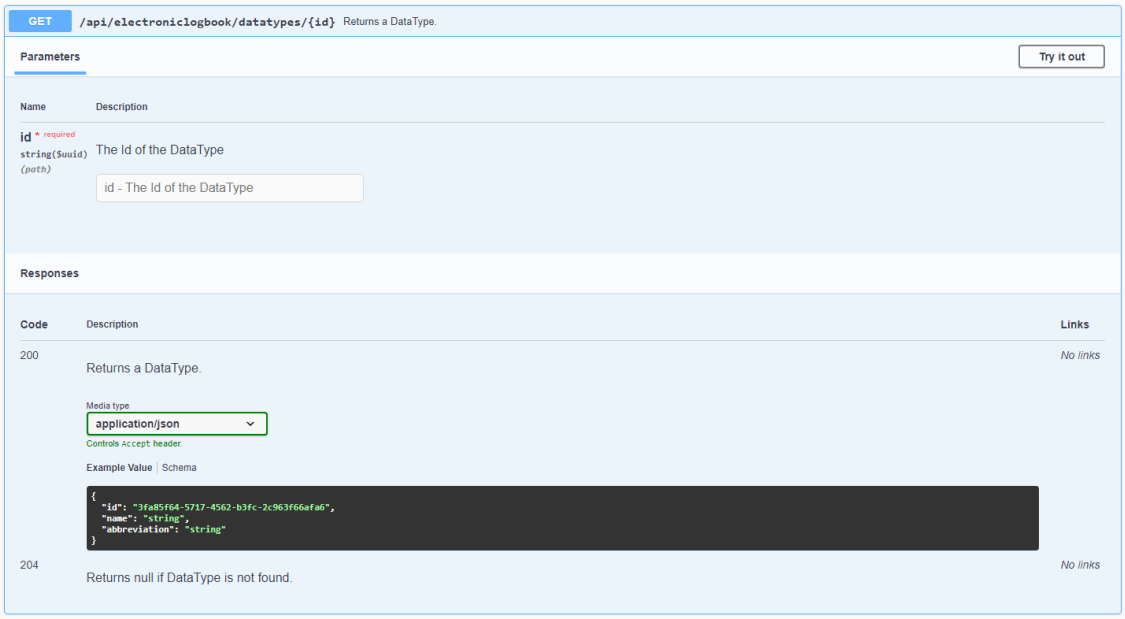


Abbildung 3: Swashbuckle Dokumentation des Endpoints /api/electroniclogbook/datatypes/{id}

## 2.6 Integrationstests

Um die API auf ihre Funktion zu testen wurden für jeden Endpoint Integrationstests erstellt. Diese testen die API anhand eines bekannten Stands in der Datenbank. Für die Tests wurde das Unittest-Tool xUnit genutzt (vgl. Wilson, et al., 2021). Für jeden Rückgabetyp gibt es einen eigenen Test. Im folgenden Codeteil sieht man einen Test für den Endpoint `/api/electroniclogbook/datatypes/{id}`.

```
[Theory]
[InlineData("FC377813-41A3-48EB-9892-284196993BF1",
            "System.Int32", "int")]
[InlineData("CB32DAB7-71DB-4303-82E5-3AEA1DC5E840",
            "System.String", "string")]
[InlineData("7307C067-4A47-4C9A-8724-48DCDA6A14DE",
            "System.DateTime", "datetime")]
[InlineData("143C6187-F278-418A-AD58-6578E218554F",
            "System.Boolean", "bool")]
[InlineData("3ADC2172-94DB-4D3D-8FF6-72CAA1103B41",
            "System.Double", "double")]
public async Task GetDataTypeTest(string typeId, string typeString,
string typeAbbreviation)
{
    var dataType = new DataType();
    await ElectronicLogBookService.GetDataTypeAsync(Guid.Parse(typeId),
            item => dataType = item).ConfigureAwait(false);
    Assert.NotNull(dataType);
    Assert.Equal(typeString, dataType.Name);
    Assert.Equal(typeAbbreviation, dataType.Abbreviation);
}
```

Codeteil 3: Integrationstest Beispiel

## 2.7 ASP.NET Core Blazor

ASP.NET Core Blazor ist ein Web-Framework, dass dem Entwickler erlaubt, Web-Applikationen mit HTML und C# zu erstellen. Blazor-Anwendungen nutzen wiederverwendbare Razor-Komponenten, die mithilfe der HTML-Syntax beliebig zusammengesetzt werden. Razor-Komponenten bestehen aus Razor-Markup, HTML und C#-Codeblöcken. Die Komponenten können Parameter haben und mithilfe von Callbacks mit ihren Elternkomponenten kommunizieren. Da Blazor in der Sprache C# geschrieben wird, können die gleichen Bibliotheken in der Webapplikation und dem Webservice verwendet werden. Der Codeteil 4 zeigt ein Beispiel zweier solcher Komponenten. Die `@page-Anweisung` der `Index Komponente` ermöglicht das Routing der Komponenten. Darunter findet sich HTML-Code, der dann gerendert wird. Hier können ebenfalls andere Razor-Komponenten aufgerufen werden, wie in diesem Beispiel die `AnotherComponent-Komponente`. Hier wird auch direkt der Wert des Attributes `Color` gesetzt. Der Inhalt dieser Komponente wird dann an dieser Stelle eingefügt. Der Codeblock der `Index-Komponente` ermöglicht, dass wenn der Button gedrückt wird, der Text „Button clicked!“ in die Konsole geschrieben wird. (vgl. Latham, et al., 2020; vgl. Microsoft Corporation, o.D.)

```
@*Index Component*@
@page "/index"

<h1>Index</h1>
<AnotherComponent Color="red"></AnotherComponent>
<button @onclick="@Click"></button>

@code {
    static void Click()
    {
        Console.WriteLine("Button clicked!");
    }
}

@*Another Component*@
<h3 style="color: @Color">AnotherComponent</h3>

@code {
    [Parameter]
    public string Color { get; set; }
}
```

Codeteil 4: Razor Komponente Beispiel

Es gibt zwei verschiedene Varianten von ASP.NET Core Blazor:

- **Blazor WebAssembly** ist ein Single-Page-Application Framework, um interaktive clientseitige Anwendungen mit Blazor zu entwickeln. Blazor WebAssembly nutzt den WebAssembly Standard (Rossberg, 2019) und funktioniert ohne Plugin in jedem modernen Browser. Die Razor-Komponenten und C#-Dateien

werden in .NET Assemblies kompiliert. Diese Assemblies werden mit der .NET Runtime vom Browser runtergeladen. Da die Größe der Anwendung dadurch ein kritischer Performance-Faktor ist, eignet sich Blazor WebAssembly für kleine Web-Anwendungen. (vgl. Latham, et al., 2020)

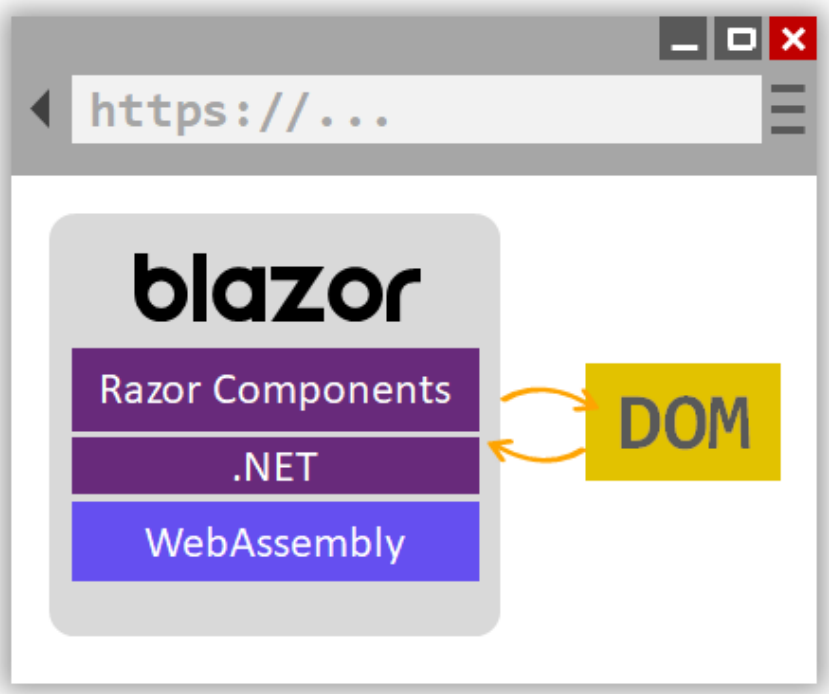


Abbildung 4: Blazor WebAssembly

- **Blazor Server** stellt Razor-Komponenten auf dem Server einer ASP.NET Core Applikation bereit. Die Benutzeroberfläche wird hierbei über eine SignalR-Verbindung aktualisiert. (vgl. Latham, et al., 2020)

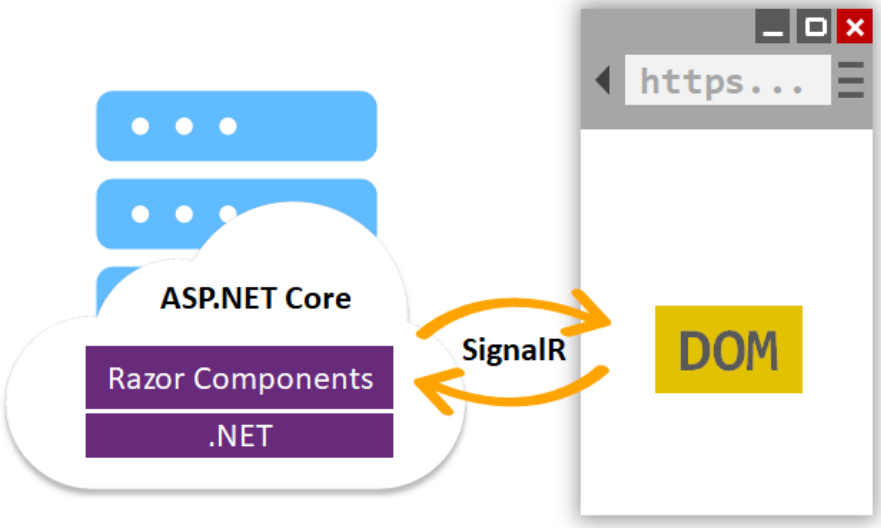


Abbildung 5: Blazor Server

Beide Varianten unterstützen die Interoperabilität mit JavaScript. Dadurch können Blazor-Anwendungen auf Browser-APIs und JavaScript-Bibliotheken zugreifen. JavaScript Funktionen können hierbei von .NET Methoden aufgerufen werden und umgekehrt.

Blazor Server hat eine schnellere Ladezeit als Blazor WebAssembly, da nicht alle Assemblies heruntergeladen werden müssen. Dafür hat die Benutzeroberfläche von Blazor WebAssembly aber eine geringere Latenz, da keine Daten über die SignalR-Verbindung hin- und hergeschickt werden müssen. Da Blazor Server eine stehende SignalR-Verbindung benötigt, ist eine offline Blazor-Server Anwendung nicht möglich.

Da beide Varianten auf Razor-Komponenten basieren, kann man jederzeit ohne großen Aufwand zum anderen wechseln.

In dieser Arbeit wird ASP.NET Core Blazor Server verwendet.

## 2.8 Telerik UI for Blazor

Die kostenpflichtigen Komponenten von Telerik stellen über 75 vorgefertigte UI-Komponenten für Blazor zur Verfügung. Diese Komponenten sind native Blazor-Komponenten. Es gibt Komponenten zur Datenverwaltung (Grid, TreeList), zur Navigation (Buttons, Pager), zur Visualisierung der Daten (Charts), zur Interaktion und User Experience (Progress Bar, Loader), zur Terminplanung (Kalender), zum Umgang mit Dokumenten (File Upload, PdfProcessing), zum Layout (Modals, Tooltips, Forms) und zum Editieren von Daten (Textboxen, DropDowns, DatePicker). (vgl. Progress Software Corporation, o.D.)

Die Telerik UI-Komponenten unterstützen sowohl Blazor WebAssembly als auch Blazor Server und werden als NuGet-Paket in das Projekt eingebunden. Um die Komponenten besser an das Projekt anzupassen, unterstützten die Komponenten mehrere Themes: das Default-Telerik-Theme, ein Theme, dass dem Bootstrap Styling entspricht und ein Theme, dass die Material Design Guidelines implementiert. Da dieses Projekt auch Bootstrap nutzt, wird hier das Bootstrap Theme verwendet.

Anfangs wurde für diese Zwecke die Bibliothek Blazorise benutzt, die ähnlich wie Telerik auch UI-Komponenten zur Verfügung stellt (vgl. Macanović, 2021). Im Gegensatz zu den kostenpflichtigen Komponenten von Telerik handelt es sich hier um Open-Source Komponenten. Im Großen und Ganzen sind die beiden Komponenten ähnlich, sodass es ohne Probleme möglich ist, dass Logbuch mit beiden Komponenten zu entwickeln. Telerik verfügt aber über eine Komponente, die es ermöglicht, Kalender zu nutzen. Diese Kalender werden in einer anderen Anwendung der Test Factory benötigt, sodass diese Anwendung umgeschrieben wurde, um die Abhängigkeiten zu verringern.

## 2.9 Design

Zusätzlich zu den Telerik UI-Komponenten wird zum einfachen Aufbau der Webseite das CSS-Framework Bootstrap verwendet. Mit Bootstrap kann man responsive Webseiten erstellen, sodass die Seiten sich an die Fenstergröße des Browsers anpassen können. (vgl. Otto, 2021)

Um das Styling mit CSS zu vereinfachen wird die Stylesheet Sprache Sass verwendet. Sass ist eine Erweiterung für CSS und erlaubt CSS verschachtelt zu schreiben, Variablen und Mixins zu nutzen. Aus dem Sass-Code kann dann wieder ein CSS-Skript generiert werden, welches in der Webanwendung genutzt werden kann. Diese Erweiterung erleichtert dem Entwickler, komplexen CSS-Code übersichtlich in verschiedenen Dateien getrennt zu schreiben. Daraus folgt ein Code, der einfacher zu lesen ist und dadurch einfacherer zu erweitern und zu warten ist. (vgl. Weizenbaum, et al., 2021)

## 2.10 NuGet

NuGet ist der Paketmanager für .NET. Ein NuGet-Paket besteht aus kompiliertem Code (beispielsweise Dynamic Link Librarys (DLLs)) und anderen Dateien, die benötigt werden, um das Projekt zu nutzen und eine Paketbeschreibung, die auch eine Versionsnummer enthält. NuGet-Pakete sind ZIP-Dateien mit der Dateiendung `.nupkg`. NuGet-Pakete können auf privaten und öffentlichen Quellen veröffentlicht werden. (vgl. Douglas, et al., 2019)

NuGet-Pakete können einfach über die Kommandozeile oder über eine graphische Oberfläche im Visual Studio heruntergeladen werden. Dort kann jedes Paket auch aktualisiert gehalten werden. Für jedes Projekt werden auch alle Abhängigkeiten des Paketes angezeigt.

## 2.11 ReSharper

ReSharper ist eine Erweiterung für Visual Studio. Die Erweiterung bietet dem Entwickler mehrere Funktionen. Der ReSharper analysiert durchgehen die Codequalität und findet Fehler, Warnungen und Empfehlungen schon bevor der Code kompiliert wurde. Die gefundenen Probleme werden im Code markiert. Für die meisten Fehler und Warnungen schlägt er direkt Lösungen vor. (vgl. JetBrains s.r.o., o.D.)

Des Weiteren hilft der ReSharper bei der Codebearbeitung und Codegenerierung. Der Entwickler muss beispielsweise bei der Implementierung eines Interfaces, keine Member von Hand implementieren, da der ReSharper diese Funktionalität über einen Shortcut anwenden kann. Danach kann der Member gegeben falls anpasst werden. Dem Ent-

wickler hilft die Erweiterung auch bei der Umbenennung jedes Symbols. Wenn mit dem ReSharper, beispielsweise eine Methode, ein Namespace, eine Variable umbenannt wird, werden automatisch alle Referenzen aktualisiert, sodass man nicht den ganzen Code durchsuchen muss. (vgl. JetBrains s.r.o., o.D.)

Eine weitere wichtige Funktion des ReSharpers ist die Codebereinigung und Codeformatierung. Es erlaubt, dass das Entwicklerteam einen gemeinsamen Codestil definiert und zwingt den Entwickler beispielsweise geschweifte Klammern nach jeder if-Anweisung zu schreiben. Der neu geschriebene Code kann dann automatisch an den definierten Standard angepasst werden. (vgl. JetBrains s.r.o., o.D.)

Der ReSharper bietet neben den anderen Funktionen noch einen verbesserten Unit-Test-Runner und weitere Verbesserungen. (vgl. JetBrains s.r.o., o.D.)

Die Codeanalyse des ReSharpers kann auch im Buildprozess durchgeführt werden, um die Codequalität beim Einchecken zu überprüfen. (vgl. JetBrains s.r.o., o.D.)

## 2.12 SonarQube

SonarQube ist ein Werkzeug zur Codeanalyse. Es erkennt Bugs, Sicherheitslücken und Code-Smells im Code. (vgl. SonarSource S.A, o.D.)

Bei jeder Continuous Integration analysiert der SonarQube den Code und schreibt das Resultat auf den SonarQube-Server. Mit der Visual Studio Erweiterung SonarLint erhält der Entwickler das Feedback direkt in der Entwicklerumgebung, zusammen mit Fehlern, Warnungen und Empfehlungen, die beim Schreiben entdeckt werden. Im Gegensatz zum ReSharper (siehe Kapitel 2.11) findet der SonarQube deutlich mehr Fehler, Warnungen und Empfehlungen, sodass der SonarQube nicht durch den ReSharper ersetzt werden kann.

Die vordefinierten Regeln kann man benutzerdefiniert gewichten und auch abschalten, wenn zu viele False-Positives erzeugt werden.

## 2.13 Font Awesome

Font Awesome bietet über 7800 verschiedene Icons, die einfach auf einer Webseite angezeigt werden können. Es gibt 5 verschiedene Styles: Solid, Regular, Light, Duotone und Brands. Unter den Styles Solid, Regular, Light und Duotone findet man Variationen der normalen Icons. Unter dem Style Brands findet man Icons von bekannten Marken. Die Styles Solid und Brands sind in der kostenlosen Version verfügbar. (vgl. Madole, et al., 2021)

Wie man in dem folgenden Codeteil sehen kann, können die Icons einfach, über das Setzen von Klassen, verwendet werden.

```
<i class="fas fa-user"></i>
```

Codeteil 5: Font Awesome Beispiel

In diesem Beispiel wir durch die Klasse `fas` angegeben, dass der Style Solid genutzt werden soll. Die Klasse `fa-user` gibt an, welches Icon verwendet werden soll. Die Icons haben alle den `fa-` Präfix.

# 3 Anforderungen

Dieses Kapitel stellt die funktionalen und nicht funktionalen Anforderungen des elektronischen Logbuchs dar. Das elektronische Logbuch soll dem Endnutzer erlauben, über das User Interface auf das Logbuch zuzugreifen (siehe Kapitel 4.2). Gleichzeitig sollen die einzelnen Anwendungen der Test Factory ebenfalls über die REST-API auf das Logbuch zugreifen.

Vor dieser Arbeit, lagen bereits Wireframes vor, die beschreiben, wie die Anwendung aussehen und funktionieren soll. Aus diesen Entwürfen und aus Gesprächen mit den Stakeholdern wurden die folgenden Anforderungen herausgearbeitet.

## 3.1 Funktionale Anforderungen

Dieses Unterkapitel stellt die funktionalen Anforderungen an das elektronische Logbuch dar. Das elektronische Logbuch soll sowohl von Endnutzern über ein User interface (UI) als auch von den anderen Anwendungen der Test Factory über die REST-API benutzt werden können.

Die Endanwender sind alle Anwender, die auf das elektronische Logbuch zugreifen können. Später wird es hier Anwender aus verschiedenen Gruppen mit verschiedenen Rechten geben. Diese sind dann in der Teilapplikation Administration der Test Factory hinterlegt, sodass alle anderen Applikationen diese Gruppen und Rechte verwenden können (siehe Anhang B, Abbildung 15). Da diese Applikation aber noch nicht implementiert wurde, wird dies in dieser Arbeit ignoriert.

Die anderen Anwendungen der Test Factory nutzen die REST-API, um beispielsweise neu erstellte Battery Cell Types zu loggen oder Änderungen an den Rechten und Gruppen zu protokolieren.

Die folgenden Abbildungen beschreiben die Use Cases, die für beide Akteure notwendig sind.

Abbildung 6 zeigt die Use Cases, die eintreten können, wenn der Endanwender oder eine Anwendung der Test Factory mit den Event Typen interagieren. Beide Akteure sollen neue Event Typen erstellen und damit definieren können. Diese Event Typen können Event Typ Attribute besitzen, die auch von beiden Akteuren erstellt werden können. Diese Attribute sollen die Datentypen Boolean, String, Integer, Double und DateTime unterstützen. Sofern der Event Typ noch nicht veröffentlicht wurde, sollen die Typen und Attribute editierbar sein. Beim Editieren kann ein Event Typ auch

veröffentlicht werden. Event Typen und deren Attribute, zu denen es keine Event Entries gibt, können ebenso gelöscht werden.

In der Abbildung 7 werden die Use Cases für die Event Entries beschrieben. Auch hier sollen sowohl die Endanwender als auch die einzelnen Anwendungen der Test Factory damit interagieren können. Beide Akteure sollen Event Entries erstellen und bearbeiten können. Die Event Entry Attribute werden automatisch mit ihrem Default Wert hinzugefügt, sodass die Akteure diese nur bearbeiten können. Die Entries können auch zusammen mit ihren Attributen gelöscht werden.

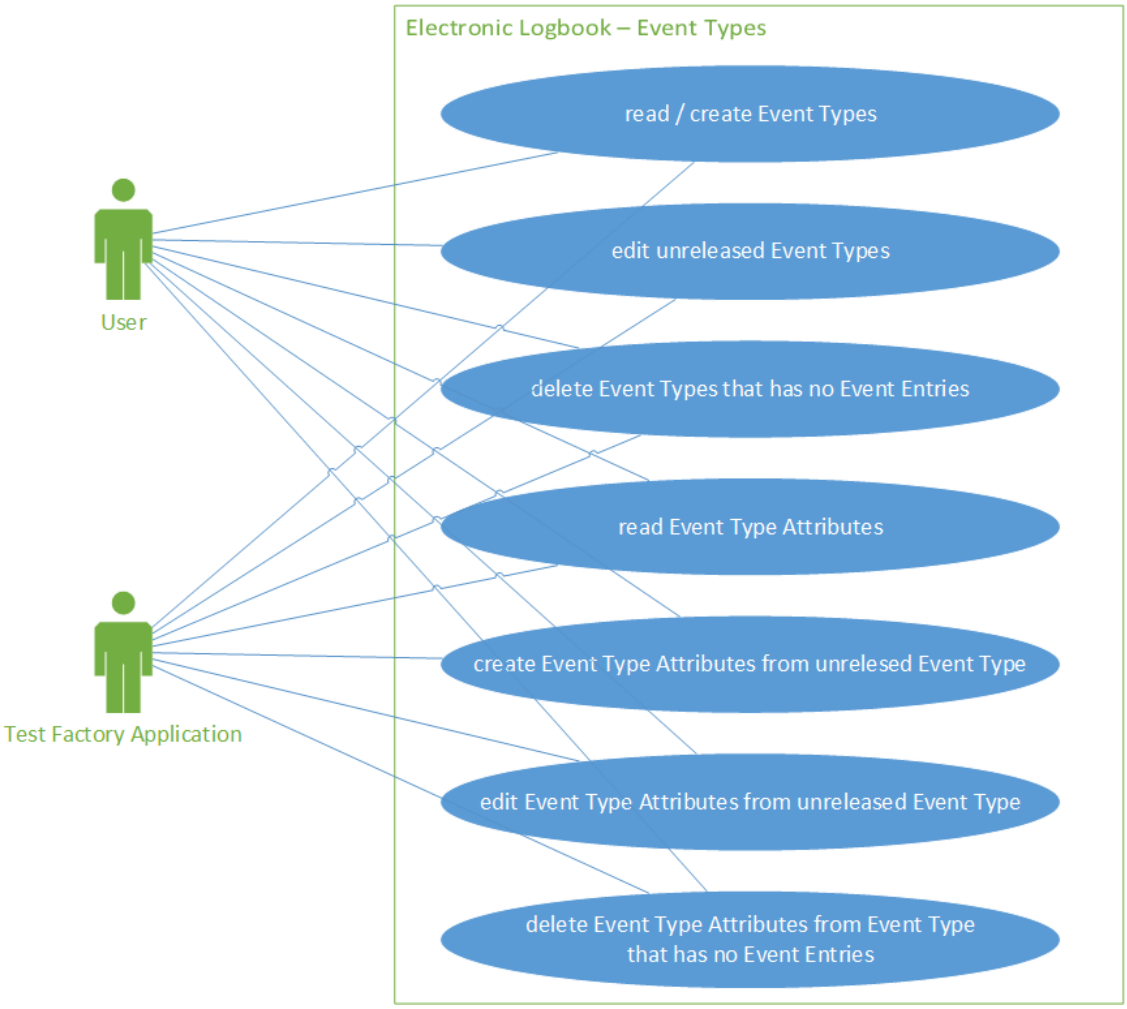


Abbildung 6: Use Cases – Event Types

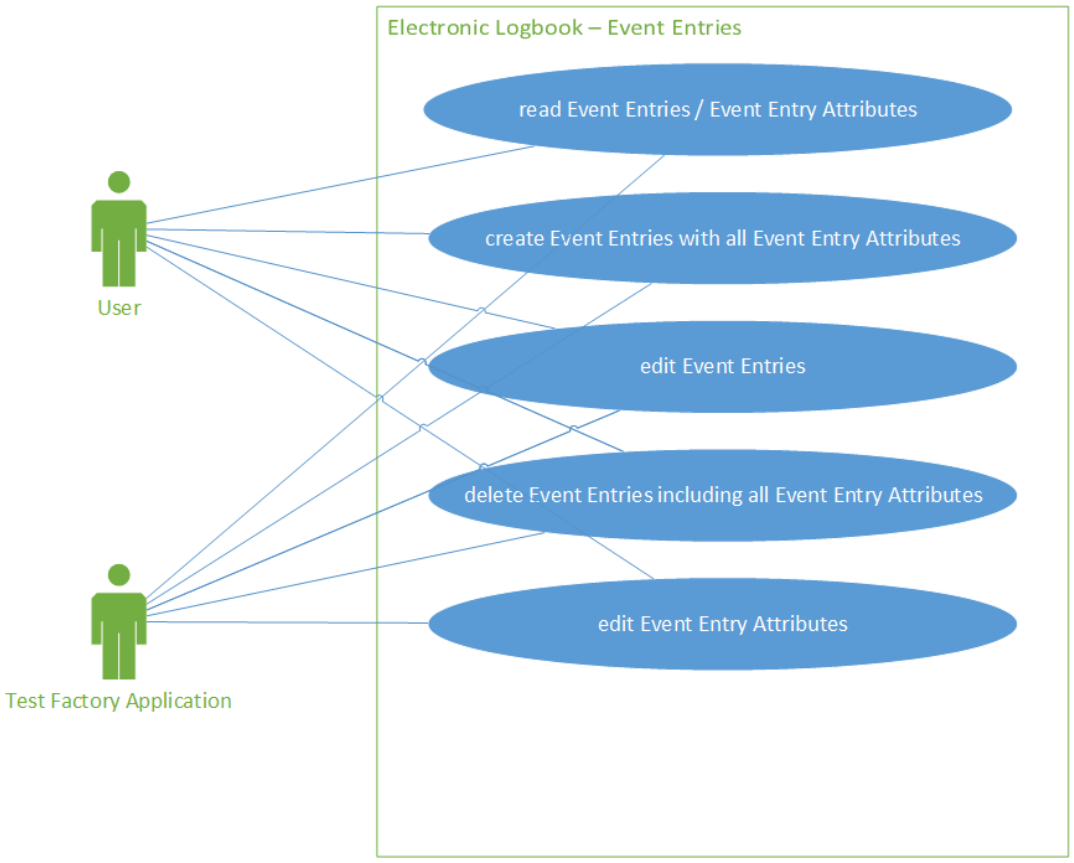


Abbildung 7: Use Cases – Event Entries

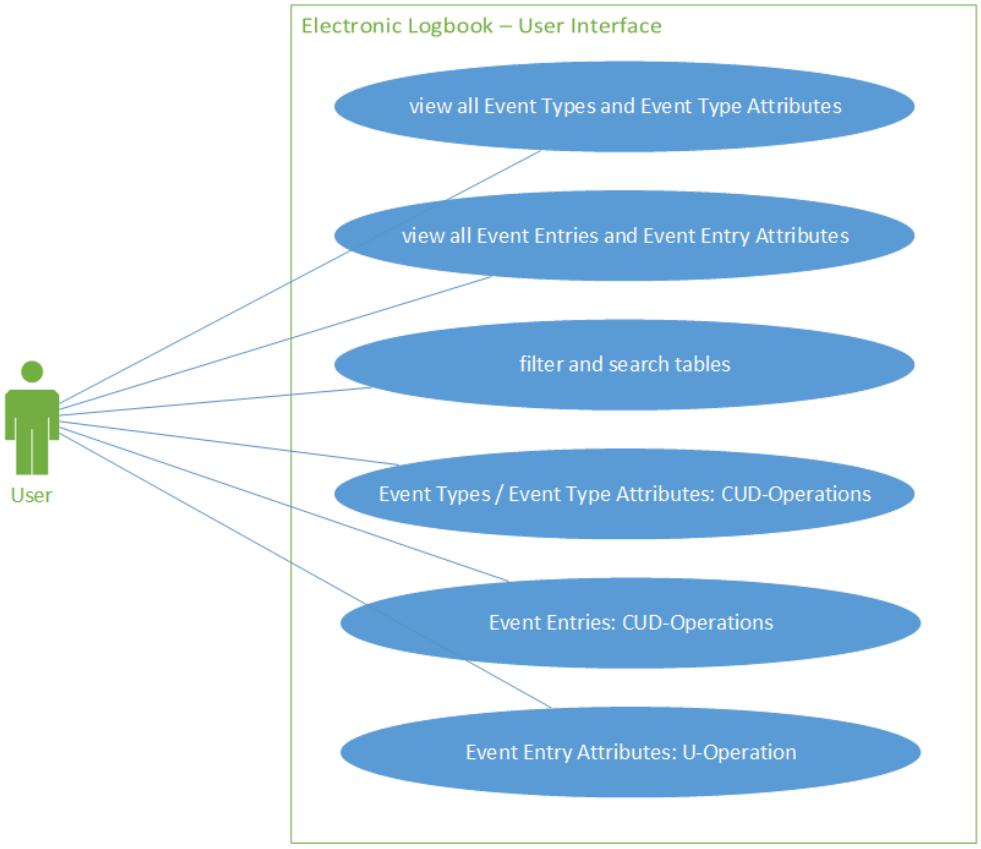


Abbildung 8: Use Cases – User Interface

Da die Endnutzer das elektronische Logbuch über das User Interface benutzen und nicht wie die einzelnen Anwendungen der Test Factory nur die REST-API benutzen, werden für die Endnutzer noch mehr Use Cases benötigt. Diese Use Cases werden in der Abbildung 8 beschrieben.

Die Benutzer sollen über das User Interface eine tabellarische Darstellung der Event Typen und Event Entries erhalten. In einer aufklappbaren Untertabelle sollen dort auch die jeweiligen Attribute angezeigt werden. Beide Tabellen sollen durchsuchbar und filterbar sein.

Um die Create (C), Update (U) und Delete (D) Operationen zu realisieren, soll der Nutzer dafür jeweils über einen modalen Dialog die Daten manipulieren können. Für jede notwendige Operation (siehe Abbildung 6 und Abbildung 7) soll ein eigenes Modal erstellt werden.

## 3.2 Nicht-funktionale Anforderungen

In diesem Unterkapitel werden die nicht-funktionalen Anforderungen des elektronischen Logbuchs beschrieben. Diese beschreiben die Anforderungen, die nicht direkt mit den Funktionen des Systems zu tun haben. Die folgenden nicht-funktionalen Anforderungen werden in verschiedenen Kategorien festgehalten.

**Wartbarkeit**

Das elektronische Logbuch soll umfangreich dokumentiert sein. Der Code muss einfach erweiterbar sein. Um Fehler zu vermeiden, soll der Code getestet werden und wird mit einer statischen Codeanalyse abgesichert.

**Benutzerfreundlichkeit**

Das User Interface des elektronischen Logbuchs muss zu den anderen Anwendungen passen. Sofern Anwendungen ähnliche Elemente besitzen, sollten diese eine identische Bedienung haben.

Die Anwendung soll eine hohe Usability aufweisen, logisch und verständlich aufgebaut sein, sodass sie einfach zu erlernen und bedienen ist. Die Benutzeroberfläche des elektronischen Logbuchs soll dabei ästhetisch aussehen.

Sollten bei der Benutzung des elektronischen Logbuchs Fehler auftreten, soll der Nutzer darüber informiert werden.

**Performance**

Jeder Request des elektronischen Logbuchs soll so schnell wie möglich laufen, sodass der Arbeitsfluss des Nutzers nicht unterbrochen wird.

### Zuverlässigkeit

Das dauerhaft korrekte Verhalten ist eine Grundvoraussetzung für die Akzeptanz des elektronischen Logbuches. Es muss zu jeder Zeit sichergestellt sein, dass das elektronische Logbuch im seltenen Fall eines Fehlers mit dem Fehler umgehen kann und so wieder zurück in einen korrekten Zustand übergehen kann.

### Technologien

Die Implementierung sollte die Software-Plattform .NET 5 nutzen. Außerdem sollen die Technologien ASP .NET Core Blazor Server, Entity Framework Core, Azure SQL Database, und die Telerik Komponenten, die im Kapitel 2 erläutert wurden, verwendet werden. Es wird die Programmiersprache C# verwendet.

Die Anwendung soll dann in Azure deployed werden.

# 4 Entwurf und Implementierung

Im vorherigen Kapitel wurden die Anforderungen an das elektronische Logbuch dargestellt. Auf Basis dieser Anforderungen zeigt dieses Kapitel, wie die Architektur und das Datenmodell des elektronischen Logbuchs aufgebaut sind und erläutert anschließend den Aufbau und die Implementierung des elektronischen Logbuchs und geht näher auf den Buildprozess und die Fehlerbehandlung ein.

## 4.1 Architektur

Bei der Architektur handelt es sich um eine Drei-Schichten-Architektur, bestehend aus einer SQL-Datenbank, einem Web Service inklusive einer REST-API und einer Webapplikation (siehe Abbildung 9).

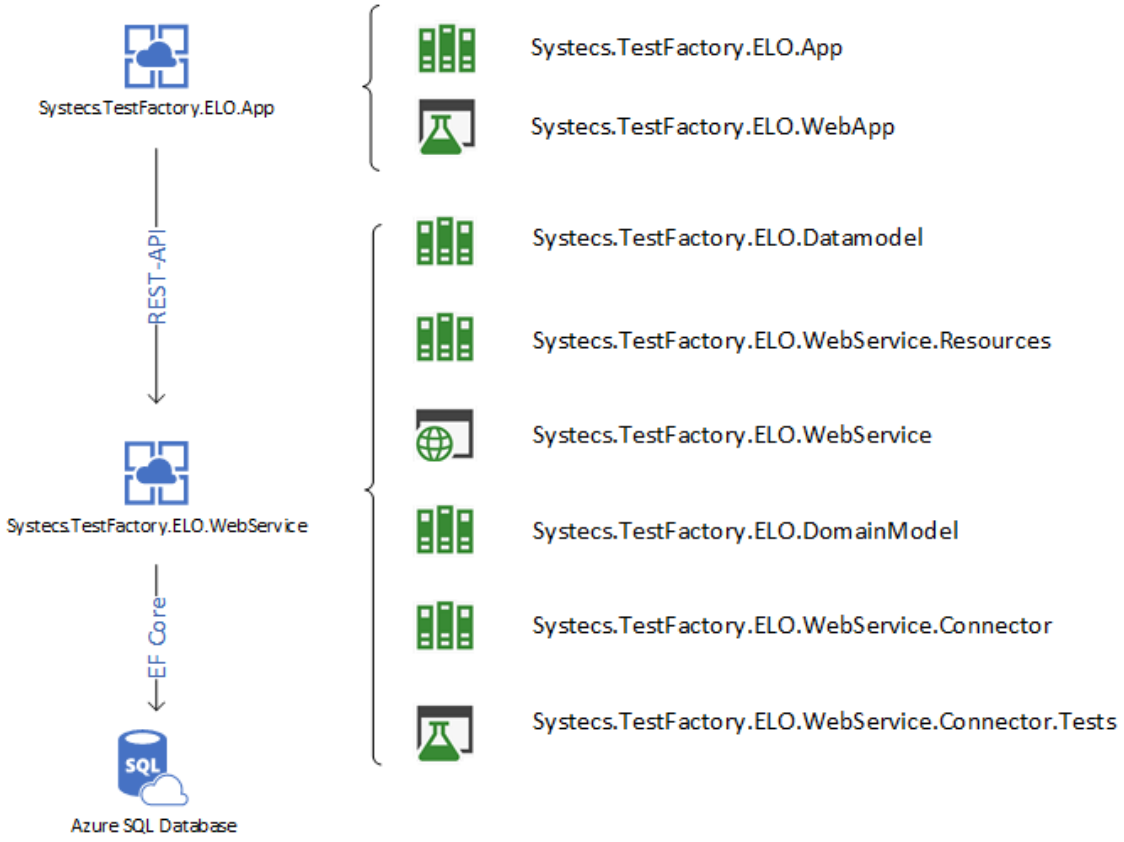


Abbildung 9: Aufbau der Architektur

Im Rahmen dieser Arbeit wurde zuerst das Datenmodell (siehe Kapitel 4.2) erstellt. Anhand des Datenmodells wurde im nächsten Schritt der Web Service implementiert. Die REST API wurde dabei mit der Datenbank getestet. Danach wurde die Webanwendung entwickelt.

## 4.2 Datenmodell

Wie man der Abbildung 10 entnehmen kann, umfasst das relationale Datenbankschema neun Tabellen, um alle funktionalen Anforderungen aus Kapitel 3 zu unterstützen. Man erkennt die Primärschlüssel, gekennzeichnet als PK und die Fremdschlüssel, gekennzeichnet als FK. Die Abbildung enthält außerdem die Datentypen und die Anzeige, ob es sich um ein NULL-Value handeln kann. Die Tabellennamen haben alle den Präfix `Elo` (Kurzform für das elektronisches Logbuch), da Tabellen von mehreren Anwendungen in der gleichen Datenbank gespeichert werden. So können gleichzeitig Namenskonflikte vermieden werden und der Entwickler sieht gleichzeitig, in welchem Kontext die Tabellen verwendet werden.

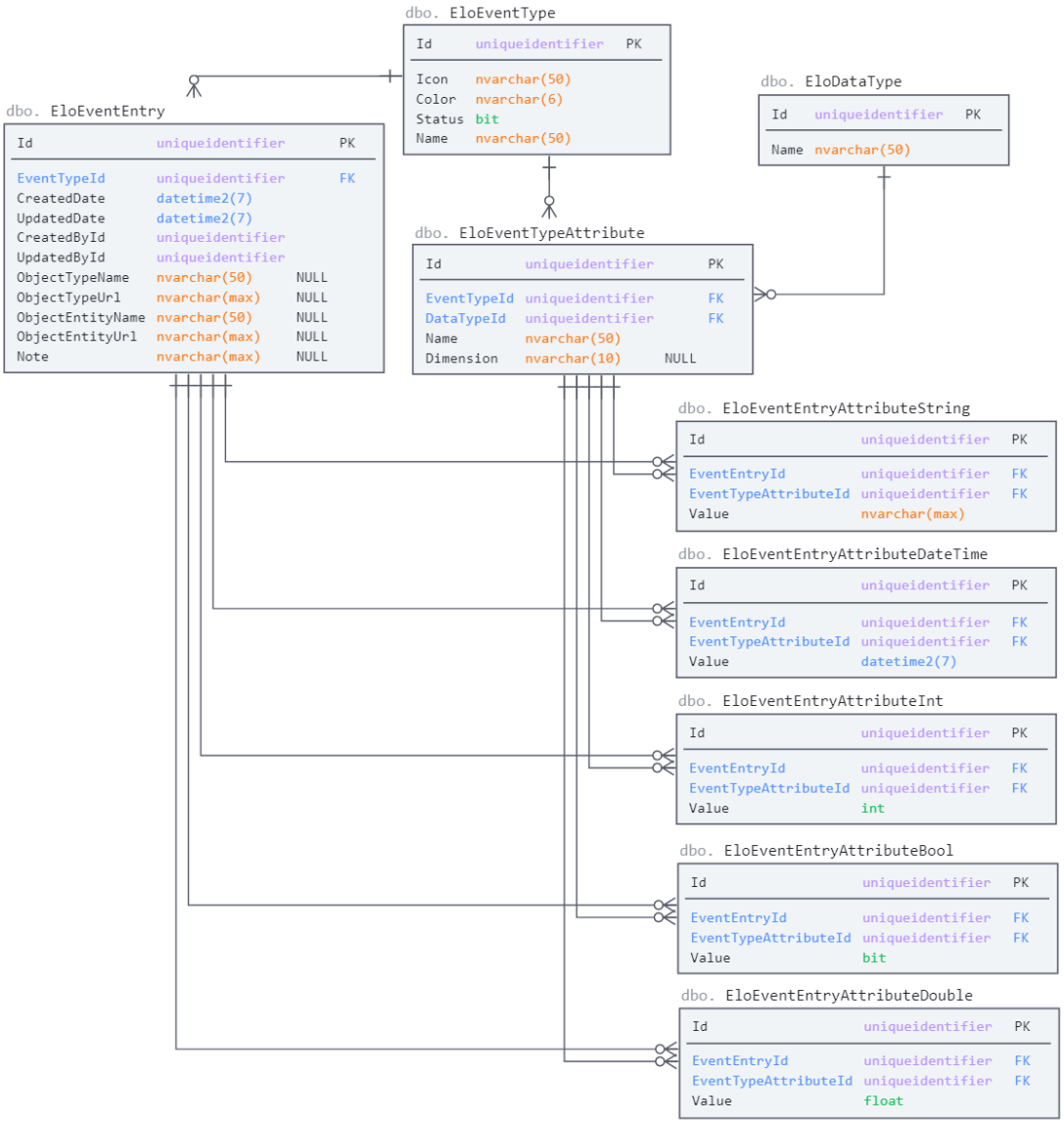


Abbildung 10: Entity-Relationship-Modell

Das Datenmodell erlaubt dem Nutzer das Erstellen generischer Event Typen und den anschließenden Eintrag von Event Entries basierend einem Event Typen. Jeder Event

Typ kann verschiedene, vom Benutzer definierte Attribute enthalten. Für diese Attribute kann der Event Entry dann Werte setzen. Dies erlaubt, dass eine Batteriezelle beispielsweise ihre Messwerte loggen kann.

In jeder Entität werden Globally Unique Identifier (GUID) verwendet, um IDs zu speichern.

In den folgenden Unterkapiteln werden die Entitäten näher erläutert.

### 4.2.1 Event Types

Die Entität `EloEventType` beinhaltet die Definition eines Event Typen. Die Spalten Icon und Color werden verwendet, dass die Typen in der Webanwendung einfacher dargestellt werden können.

Tabelle 1: Datenbanktabelle EloEventType

| **Spaltenname** | **Datentyp** | **Key, NULL** | **Kommentar** |
|---|---|---|---|
| Id | uniqueidentifier | PK | EventType-ID |
| Icon | nvarchar(50) | | Font Awesome Icon,<br>Beispiel: `fa-user` |
| Color | nvarchar(6) | | Hex-Farbcode ohne #<br>Beispiel: `FF0000` |
| Status | bit | | Gibt an ob der Event Typ veröffentlicht ist |
| Name | nvarchar(50) | | Name des Event Typen |

### 4.2.2 Data Types

Die Entität `EloDataType` enthält eine Übersicht der Datentypen, die in den Attributen verwendet werden können. Aktuell werden die Datentypen Boolean, String, Integer, Double und DateTime unterstützt.

Tabelle 2: Datenbanktabelle EloDataType

| **Spaltenname** | **Datentyp** | **Key, NULL** | **Kommentar** |
|---|---|---|---|
| Id | uniqueidentifier | PK | DataType-ID |
| Name | nvarchar(50) | | Name des .NET Datentypen<br>Beispiel: `System.Boolean` |

### 4.2.3 Event Type Attributes

Die Attribute der Event Typen werden in der Tabelle `EloEventTypeAttribute` abgelegt. Hier kann der Nutzer Attribute definieren, die an einem Event Typen hängen. Jedes Attribut nimmt einen Datentyp aus der `EloDataType` Entität an. Der Nutzer kann auch eine optionale Einheit angeben, wenn es sich beispielsweise um einen Messwert handelt. Diese Einheit wird im Moment noch als `string` abgespeichert. Später soll hier eine eigene Entität erstellt werden, sodass auch die anderen Applikationen der Test Factory die gleichen Einheiten nutzen können. Dann wird aus dieser Spalte ein Fremdschlüssel, um auf die andere Tabelle zuzugreifen. Bis dahin wird diese Lösung als Workaround dienen.

Tabelle 3: Datenbanktabelle EloEventTypeAttribute

| **Spaltenname** | **Datentyp** | **Key, NULL** | **Kommentar** |
|---|---|---|---|
| Id | uniqueidentifier | PK | EventTypeAttribute- ID |
| EventTypeId | uniqueidentifier | FK | EventType-ID |
| DataTypeId | uniqueidentifier | FK | DataType-ID |
| Name | nvarchar(50) | | Name des Attributs |
| Dimension | nvarchar(10) | NULL | Einheit des Attributs |

### 4.2.4 Event Entries

In der Entität `EloEventEntry` werden die Einträge der Event Typen gespeichert. Sie verweist auf einen Event Typen.

Die Spalten `CreatedById` und `UpdatedById` sollen abspeichern, welcher Nutzer oder welche Anwendung den Event Entry erstellt beziehungsweise zuletzt geändert hat. Da diese Entität zur Zeit der Implementierung noch nicht vorhanden war, wird dort im Moment eine GUID abgespeichert, die noch nichts aussagt. Hier kann in Zukunft dann ein Fremdschlüssel auf eine solche Tabelle gespeichert werden.

Wenn der Event Entry von einer Anwendung erstellt wird, kann in den Spalten `ObjectTypeName`, `ObjectTypeUrl`, `ObjectEntityName` und `ObjectEntityUrl` abgespeichert werden, um welches Objekt es sich handelt. Da eine solche Anwendung zur Zeit der Implementierung noch nicht fertiggestellt war, sagen diese Werte ebenfalls noch nichts aus und das Verhalten konnte, im Zusammenhang mit anderen Applikationen, noch nicht getestet werden.

Tabelle 4: Datenbanktabelle EloEventEntry

| **Spaltenname** | **Datentyp** | **Key, NULL** | **Kommentar** |
|---|---|---|---|
| Id | uniqueidentifier | PK | EventEntry-ID |
| EventTypeId | uniqueidentifier | FK | EventType-ID |
| CreatedDate | datetime2(7) | | Zeitstempel der Erstellung |
| CreatedById | uniqueidentifier | | User-ID oder Anwendungs-ID des Erstellers |
| UpdatedDate | datetime2(7) | | Zeitstempel der letzten Aktualisierung |
| UpdatedById | uniqueidentifier | | ID des Users oder der Anwendung, der den Event Typen zuletzt aktualisiert hat |
| ObjectTypeName | nvarchar(50) | NULL | Name des Objekt Typen<br>Beispiel: Ionenbatteriezelle |
| ObjectTypeUrl | nvarchar(MAX) | NULL | URL zur Definition des Objekt Typen |
| ObjectEntityName | nvarchar(50) | NULL | Name des Objekt Entität<br>Beispiel: Zelle 1 |
| ObjectEntityUrl | nvarchar(MAX) | NULL | URL zur Definition der Objekt Entität |
| Note | nvarchar(MAX) | NULL | Notiz |

### 4.2.5 Event Entry Attributes

Um die Werte der definierten EventTypeAttribute protokollieren zu können, benötigt es eine Entität, in der diese Werte abgespeichert werden. Hier benötigt man eine Spalte, in der man einen Wert abspeichern soll, der verschiedene Datentypen annehmen kann.

Um dieses Problem zu lösen, gibt es mehrere Möglichkeiten:

- den Wert als `string` umwandeln und den String abspeichern,
- den Datentyp `sql_variant` benutzen,
- den Datentyp `xml` benutzen,
- den Wert als `JSON` zusammen mit dem Datentyp abspeichern,
- für jeden Datentyp eine eigene Spalte erstellen oder

- für jeden Datentyp eine eigene Entität erstellen.

Wenn man den Wert als `string` abspeichert, hat man das Problem, dass wenn die Typumwandlung fehlerhaft ist, der Wert nicht mehr zurückkonvertierbar ist. Der Datentyp `sql_variant` gibt in EF Core ein Objekt von Typ `object` zurück, sodass man dort ebenfalls den Typ umwandeln muss. Wenn man für jeden Datentyp eine eigene Spalte erstellt, hat die Datenbank viele NULL-Werte.

Hier wurde entschieden, für jeden Datentyp eine neue Tabelle zu erstellen. Es ist nicht geplant, dass mehr Datentypen in Zukunft unterstützt werden, sollte dies aber der Fall sein, muss nur eine neue Tabelle erstellt werden und im Web Service ein Interface implementiert werden, dass dem Service erlaubt, mit generischen Methoden auf die Datenbank zuzugreifen (siehe Kapitel 4.3.3). Diese Lösung erlaubt es, dass die Werte in dem dafür vorgesehenen Datentyp abgespeichert werden können. Somit können Fehler bei der Typumwandlung der Werte ausgeschlossen werden. Diese Lösung ist ebenfalls die beste Lösung, um den benötigten Speicherplatz zu minimieren.

Es wäre aber genauso möglich, den Wert als `JSON` oder `xml` abzuspeichern und dann dies im Service auszulesen. Durch die Schichtenarchitektur ist dies ohne großen Aufwand austauschbar, da man nur die Datenbank und die einzelnen Stellen im Web Service austauschen muss.

Die fünf Entitäten für die fünf Datentypen heißen:

- `EloEventEntryAttributeBool`,
- `EloEventEntryAttributeDateTime`,
- `EloEventEntryAttributeDouble`,
- `EloEventEntryAttributeInt` und
- `EloEventEntryAttributeString`.

Tabelle 5: Datenbanktabelle EloEventEntryAttribute

| **Spaltenname** | **Datentyp** | **Key, NULL** | **Kommentar** |
|---|---|---|---|
| Id | uniqueidentifier | PK | EventEntryAttribute-ID |
| EventEntryId | uniqueidentifier | FK | EventType-ID |
| EventTypeAttributeId | uniqueidentifier | FK | EventTypeAttribute-ID |
| Value | bit / datetime2(7) / float / int / nvarchar(MAX) | | Wert des Attributes |

### 4.2.6 Performance der Datenbank

Um die Performance der Datenbank zu testen, wird eine Testdatenbank mit generierten Daten aufgesetzt. Es werden im ersten Test noch keine zusätzlichen Indizes erstellt, um die Performance zu erhöhen. Über ein SQL-Skript (siehe Anhang A.3, Codeteil 16) werden automatisch Testdaten generiert.

Mit dem Skript werden in diesem Fall 100 Event Typen erstellt mit jeweils einem Event Typ Attribut für jeden unterstützen Datentyp (≙ 5 * 100 Attribute). Zusätzlich werden für jeden Event Typen 10.000 Event Entries (≙ 1.000.000 Event Entries), inclusive deren Event Entry Attributen erstellt (≙ 5 * 1.000.000).

Anhand dieser Testdaten wird mit einem weiteren SQL-Skript (siehe Anhang A.3, Codeteil 17) ein zufälliges Event Entry und die Werte der dazugehörigen Attribute ausgelesen. Das Skript misst die Ausführungszeit. Das Ergebnis wird in der folgenden Abbildung dargestellt.

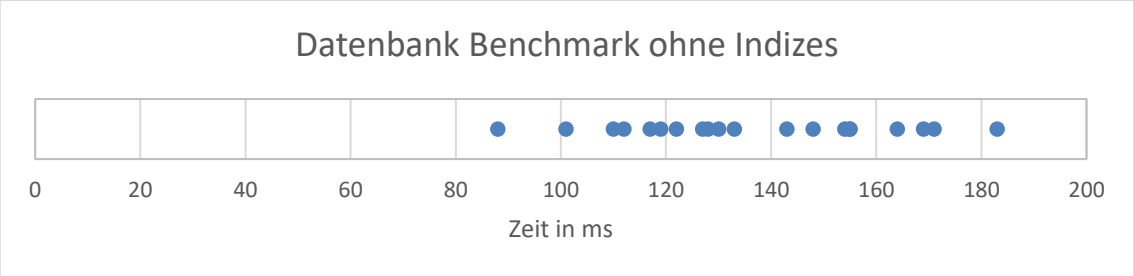


Abbildung 11: Datenbank Benchmark ohne Indizes

Der arithmetische Mittelwert der Ausführungszeit dieser Query beträgt hier nach 20 Durchläufen 136 ms.

Sofern man jeweils einen Index auf die Spalte EventEntryId der Event Entry Attribute hinzufügt, die Spalte, die hier für die Join-Operation benutzt wird, kann die Ausführungszeit halbiert werden. Dies wird in der folgenden Abbildung dargestellt.

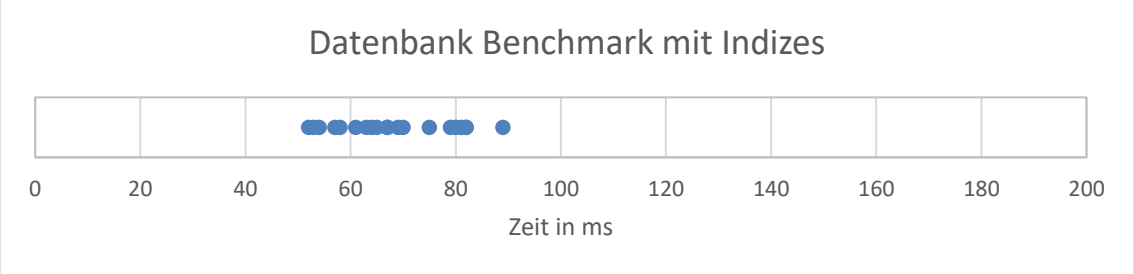


Abbildung 12: Datenbank Benchmark mit Indizes

Der arithmetische Mittelwert der Ausführungszeit der Query beträgt jetzt 68 ms. Wenn der Eintrag in der Datenbank bereits gecached ist, beträgt die Ausführungszeit weniger

als eine Millisekunde. Somit stellt diese Implementierung aus Performancegründen kein Problem dar.

## 4.3 Web Service

Der Web Service wird in sechs Projekte unterteilt, die im folgenden Kapitel näher beschrieben werden. Er enthält die Entitätsobjekte, die Datentransferobjekte und die REST-API.

### 4.3.1 Datamodel

Die Klassenbibliothek Datamodel besteht aus den mit EF Core generierten Entitätsobjekten der Datenbank (siehe Kapitel 2.3). Im folgenden Codeteil sieht man das Skript, dass verwendet wurde, um aus der Datenbank die Klassen zu generieren. Das Skript wird in der Package Manager Console (PMC) ausgeführt.

```
Scaffold-DbContext "Server=localhost;Database=ELO-DEV;
Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer
-Context EloDbContext -force -v -d -StartupProject
Systecs.ELO.Datamodel -Project Systecs.ELO.Datamodel
```

Codeteil 6: Scaffold-DbContext PMC-Skript

Bei den generierten Klassen handelt es sich um partielle Klassen. Dadurch sind die Entitätsobjekte erweiterbar, sodass mögliche Änderungen nach dem nächsten Reverse-Engineering rückgängig nicht gemacht werden. Dieses Feature wird genutzt, um den generischen Zugriff auf die verschiedenen EventEntryAttribute-Tabellen möglich zu machen.

```
public interface IEventEntryAttribute<T>
{
    public Guid Id { get; set; }
    public Guid EventEntryId { get; set; }
    public Guid EventTypeAttributeId { get; set; }
    public EloEventEntry EventEntry { get; set; }
    public EloEventTypeAttribute EventTypeAttribute { get; set; }
    public T Value { get; set; }
}

public partial class EloEventEntryAttributeBool :
      IEventEntryAttribute<bool> {};
public partial class EloEventEntryAttributeDateTime :
      IEventEntryAttribute<DateTime> {};
public partial class EloEventEntryAttributeInt :
      IEventEntryAttribute<int> {};
public partial class EloEventEntryAttributeString :
      IEventEntryAttribute<string> {};
public partial class EloEventEntryAttributeDouble :
      IEventEntryAttribute<double> {};
```

Codeteil 7: Interface IEventEntryAttribute<T>

So kann der Web Service anhand dem Datentyp des Attributes die richtige Tabelle ansteuern (siehe Kapitel 4.3.3).

### 4.3.2 Domainmodel

Die Datentransferobjekte werden in der Klassenbibliothek Domainmodel hinterlegt. Da diese Objekte über die REST-API übergeben werden, implementieren sie das `ISerializable` Interface. Für Datentransferobjekte, die in der UI manipuliert werden können, werden hier `DataAnnotationsAttribute` vor den jeweiligen Properties hinterlegt. So kann in der UI eine einfache Validierung der Properties aufgrund der Attribute durchgeführt werden (siehe Kapitel 4.4.2). Dies ist im Codeteil 14 im Anhang A.1 anhand des Datentransferobjektes `EventType` ersichtlich.

### 4.3.3 WebService

Das Projekt WebService besteht aus der REST-API und ihrer Dokumentation (siehe Kapitel 2.5), sowie den Mapping-Definitionen (siehe Kapitel 2.4) und sonstiger Geschäftslogik. Zur besseren Übersicht ist das Projekt in folgende Namespaces unterteilt:

- Controller
- Mapping
- Services
- Logging

Im Namespace Controller befinden sich die Einstiegspunkte der REST-API (siehe Codeteil 2). Dort sind ebenfalls die XML-Kommentare und die Attribute für die

Swashbuckle Dokumentation (siehe Kapitel 2.5) zu finden. Jede Methode ruft dann ihre Namensgleiche Methode im Namespace Services auf.

Der Namespace Mapping enthält die Mapping-Definitionen, um die Datentransferobjekte in Entitätsobjekte umzuwandeln und umgekehrt (siehe Kapitel 2.4). Diese Definitionen werden am Start des Projektes in der Methode `ConfigureServices()`, der Startup Klasse, einmalig aufgerufen.

Der Namespace Logging enthält Hilfsklassen, die das Logging der REST-API auf die Konsole erleichtern. Somit ist einfaches Debugging auch während dem Betrieb möglich, da man diese Logs in Azure auslesen kann. Bisher loggt jede Klasse den Start und das Ende der jeweiligen Methode im Service. Außerdem wird vor und nach dem Aufrufen der `DbContext.SaveChangesAsync()` Methode ebenfalls geloggt, da dort erfahrungsgemäß die meisten Fehler passieren. Diese Methode speichert die Änderung in die SQL-Datenbank. Sollte während der Service-Methode ein bekannter Fehler, wie zum Beispiel ein Namenskonflikt, auftreten, wird dieser ebenfalls geloggt.

Im Namespace Services befinden sich die Methoden, die für die REST-API mit der Datenbank kommunizieren und die Daten anschließend, wenn nötig, überprüft und in Datentransferobjekte umwandelt. Neben mehrerer Hilfsklassen wird hier auch der generische Zugriff auf die verschiedenen EventEntryAttribute-Tabellen geregelt.

```
public async Task<EloActionResult<IList<EventTypeDto>>>
      GetEventTypesAsync()
{
    using var scope = new AutoMethodScopeLogger(_logger);

    var entities = await DbContext.EloEventTypes
                                  .ToListAsync()
                                  .ConfigureAwait(false);

    return Ok(_mapper.Map<IList<EventTypeDto>>(entities));
}
```

Codeteil 8: Methode GetEventTypesAsync()

Der Codeteil 8 zeigt die Methode `GetEventTypesAsync()`. Hier wird am Anfang der Methode das Logging über die Hilfsklasse `AutoMethodScopeLogger`, die im Namespace Logging zu finden ist, definiert. Der `AutoMethodScopeLogger` implementiert das IDisposable Interface, sodass am Anfang im Konstruktor-Aufruf der Start der `GetEventTypesAsync()` Methode geloggt wird. Das Interface IDisposable bringt die Methode `Dispose()` mit, die am Ende des Scopes, also am Ende der Methode `GetEventTypesAsync()`, aufgerufen wird. Hier wird dann das Ende der Methode geloggt. Zuvor werden alle EventTypes aus der Datenbank besorgt, in Datentransferobjekte umgewandelt und zurückgegeben.

Wenn man alle Event Entry Attribute eines Event Entries besorgen möchte, benötigt man eine kompliziertere Query. Hier soll es möglich sein, auf alle Attribute generisch zuzugreifen, ohne jeden Zugriff auf eine der Datenbank-Tabellen im Code hart zu kodieren.

Zuerst wurde dies Mittels Reflection gelöst. Dies ist aus Performancegründen aber nicht zu gebrauchen, da ein Aufruf einer Methode über eine Reflection um ein Vielfaches langsamer ist als ein normaler Methodenaufruf (vgl. Warren, 2016). Der Codeteil 15 im Anhang A.2 zeigt die aktuelle Implementierung am Beispiel der Methode `GetEventEntryAttributesAsync(Guid eventEntryId)`. Hier wird sich am Anfang der Methode das Dictionary `EventEntryAttributeInstanceHelper-Dictionary` besorgt. Dieses Dictionary besteht aus dem Defaultwert des Datentyps (zum Beispiel, `false`) und der dazugehörigen Klasse des Entitätsobjektes (zum Beispiel `EloEventEntryAttributeBool`).

Mit diesem Workaround kann der Entwickler einfach neue Datentypen hinzufügen, indem er zuerst die neue Tabelle der Datenbank hinzufügt, der danach neu generierten Entitätsklasse das richtige Interface implementieren lässt (siehe Codeteil 7) und anschließend den Eintrag im Dictionary hinzufügt.

Zur Zeit der Ausarbeitung dieser Arbeit wird diskutiert, ob dieses Verhalten auch mit einem Visitor Pattern erreicht werden kann. Sollte dies erwünscht sein, wird sich der Code an dieser Stelle angepasst.

### 4.3.4 WebService.Ressources

Die Klassenbibliothek WebService.Ressources besteht aus Ressource-Dateien. Diese werden unter anderem für die REST-API benötigt, um bei vordefinierten Fehlern eine Fehlermeldung mitzugeben, die dann auch in der UI angezeigt werden kann. Sofern sonstige Dateien gebraucht werden, können diese ebenfalls in dieser Bibliothek abgelegt werden.

### 4.3.5 Connector

Der Connector enthält Wrapper-Methoden zum einfachen Aufruf der REST-API in C#. So muss der Entwickler in Blazor nicht jedes Mal einen Rest Request erstellen, sondern kann direkt diese Methoden benutzen. In allen anderen Sprachen ist die REST-API normal über die definierten Endpoints erreichbar.

```
public Task GetEventTypesAsync(Action<IList<EventType>> OnSuccess)
{
    return GetEventTypesAsync(OnSuccess, null);
}

public async Task GetEventTypesAsync(Action<IList<EventType>>
     OnSuccess, Action<EloRequestFailed> OnError)
{
    var request = new RestRequest(new Uri($"{ApiEventTypesPath}",
      UriKind.Relative));
    await _connector.ExecuteGetAsync(request, OnSuccess,
      OnError).ConfigureAwait(false);
}
```

Codeteil 9: Connector-Methode GetEventTypesAsync()

Für jeden API Endpoint gibt es hier zwei Methoden. Eine Methode mit einem `OnSuccess` Callback und eine mit jeweils einem `OnSuccess` und `OnError` Callback. Der `OnSuccess` Callback gibt im Erfolgsfall die gewünschte Antwort zurück. Sofern innerhalb der REST-API ein Fehler unterlaufen ist, gibt der `OnError` Callback ein Objekt der Klasse `EloRequestFailed` zurück. Dort ist unteranderem der HTTP-Fehlercode und eine Beschreibung des Fehlers enthalten. Wenn die überladende Methode aufgerufen wird, die keinen `OnError` Callback hat, wird im Fehlerfall eine Exception geworfen.

Aus dieser Bibliothek wird zusammen mit der Projektreferenz der Domainmodels ein NuGet Paket erstellt. Dieses Paket kann dann in der App eingebunden und benutzt werden. So erhält die App gleichzeitig alle Domainmodels und Methoden zum Aufrufen der REST-API. Das NuGet Paket wird automatisch im Buildprozess erstellt (siehe Kapitel 4.5).

### 4.3.6 Connector.Test

Im Projekt Connector.Test sind die Integrationstests des Connectors zu finden (siehe Kapitel 2.6). Hier wird jede Connector-Methode anhand eines bekannten Stands in der Datenbank getestet. Für Methoden, in der vordefinierte Fehler auftreten können, gibt es mehrere Tests, um auch die Funktionalität dieser Fehler zu testen.

Im folgenden Codeteil kann man ein solches Beispiel sehen. Hier wird versucht ein neuen EventType anzulegen mit einem bereits verwendeten Namen. In der Servicemethode des WebService wird auf diesen Fall geprüft. Die API antwortet darauf mit einem 409 Conflict. Die Connector Methode erstellt daraus das `EloRequestFailed` Objekt, welches in diesem Test überprüft wird.

```
[Fact]
public async Task CreateEventTypeTest_DuplicateName()
{
    var eventTypeToCreate = DummyEventType();
    eventTypeToCreate.Name = EventType_Name_Released;
    EloRequestFailed failedRequest = null;
     await ElectronicLogBookService
           .SaveEventTypeAsync(eventTypeToCreate, _ => { },
                 item => failedRequest = item)
           .ConfigureAwait(false);

    Assert.Equal(Exception_NameAlreadyExists,
      failedRequest.ResponseMessageKey);
    Assert.Equal(HttpStatusCode.Conflict,
      failedRequest.Response.StatusCode);
}
```

Codeteil 10: Integrationstest für den Fehlerfall: Erstellen eines EventTypes mit einem bereits existierenden Namen

## 4.4 App

Die Haupt-Webanwendung der Test Factory besteht aus mehreren Komponenten. Jede Teilanwendung stellt der Test Factory ihre Bibliothek als NuGet Paket zur Verfügung.

Um diesen Aufbau zu ermöglichen, ist die App jeder Teilanwendung in zwei Projekte unterteilt:

- eine Razor Bibliothek, die alle Razor Komponenten, JavaScript-, C#- und CSS-Dateien enthält. Diese Bibliothek wird später als NuGet Paket in die Hauptanwendung eingebunden.
- Ein einfaches Blazor-Server-Projekt, in der die Razor Bibliothek eingebunden ist, um die Anwendung lokal zu testen.

Dieses Kapitel beschreibt, wie die Webanwendung des elektronischen Logbuchs aufgebaut und implementiert ist.

### 4.4.1 App Content Frame

Um das Design der Webanwendung einheitlich zu gestalten, werden Teile der Webseite, die alle Komponenten enthalten, in eine eigene Bibliothek namens `AppContentFrame` ausgelagert. Diese Bibliothek wird parallel außerhalb des Scopes dieser Arbeit entwickelt.

Das `AppContentFrame` (siehe Abbildung 13, orangener Rahmen) ermöglicht es jeder Komponente, eine Navigationsleiste (siehe Abbildung 13, grüner Rahmen), Suchleiste und Buttons zum Erstellen und Löschen von Elementen einzufügen. Gleichzeitig kann der Titel des Fensters angepasst werden (siehe Abbildung 13, pinker Rahmen). Das

`AppContentFrame` bietet außerdem die Möglichkeit, einheitliche Nachrichten (auch als Toast oder Snackbar bekannt) anzuzeigen, nachdem man mit der Webseite interagiert hat (siehe Abbildung 14). Der Content der Teilanwendung kann dann dort einheitlich angezeigt werden (siehe Abbildung 13, roter Rahmen).

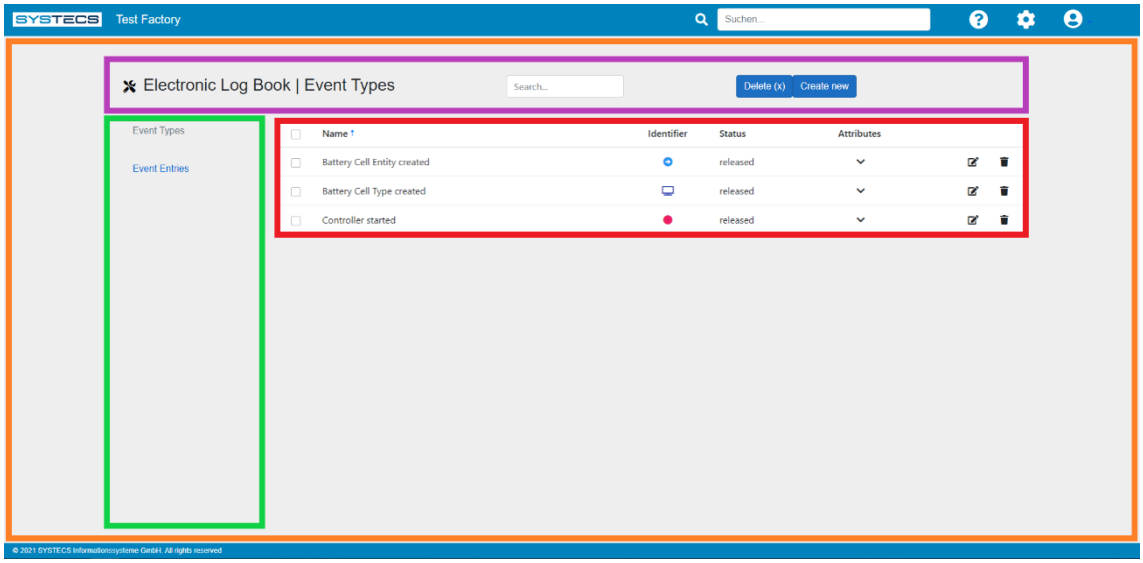


Abbildung 13: Visualisierung des AppContentFrame

Sofern ein Button oder die Suchleiste nicht benötigt wird, kann diese auch weggelassen werden.

Die Konfiguration des `AppContentFrame` erfolgt über die Implementierung eines Interfaces. Hier werden alle Sektionen der Navigationsleiste angegeben. Ebenso kann durch die Implementierung eines weiteren Interfaces angegeben werden, welche Services die Webanwendung beim Start registrieren soll, dass diese über Dependency Injection genutzt werden können.

### 4.4.2 Event Types

Um die Event Typen sauber darzustellen, gibt es dafür eine eigene Seite in der Webanwendung des elektronischen Logbuchs. Die Seite besteht hauptsächlich aus einem Telerik Grid, um die Daten in tabellarischer Form darzustellen und mehreren Modals, um die Daten zu manipulieren (siehe Anhang B, Abbildung 16).

Die Tabelle besteht aus mehreren Spalten, um die Eigenschaften des Event Typen abzubilden. Zusätzlich kann man über die Buttons das dazugehörige Modal (zum Erstellen, Bearbeiten, Löschen) öffnen. Über die Spalte Attributes kann der Nutzer unter der Reihe eine weitere Tabelle aufklappen, in der die Attribute des Event Typen visualisiert werden. Diese Untertabelle besitzt ebenfalls Buttons, um Modals zum Manipulieren der Attribute zu öffnen.

Über die Suchleiste des `AppContentFrame` kann das Grid durchsucht werden, sodass der Nutzer die Tabelle filtern kann. Über die Selektions-Box in der ersten Spalte kann der Nutzer mehrere Zeilen auswählen und kann mit dieser Auswahl über den Löschbutton des `AppContentFrame` mehrere Event Typen gleichzeitig löschen.

Die Modals sind alle gleich aufgebaut, indem sie als Basis eine gemeinsame Modal-Komponente haben. Da die Modals zum Erstellen und Bearbeiten der Event Typen die gleichen Eingabefenster haben, wurde diese Form ebenfalls in eine eigene Razor Komponente ausgelagert. Diesen Aufbau findet man auch bei den anderen folgenden Modals der Event Entry Attribute, der Event Entries und der Event Entry Attribute.

Diese Razor Komponenten können analog zu den HTML-Tags, mit einem Tag verwendet werden.

Für die Eingabeformulare in den Modals wird die Telerik Form benutzt (siehe Anhang B, Abbildung 17). Dies erlaubt dem Entwickler das einheitliche Darstellen eines Formulars. Gleichzeitig kann jedes Feld auf Basis der `DataAnnotationsAttribute` des Domainmodels validiert werden. Sofern die Validierung fehlschlägt, wird der Button zum Speichern deaktiviert und unter dem Feld wird die Fehlernachricht des `DataAnnotationsAttribute` angezeigt (siehe Anhang A.1, Codeteil 14).

Telerik bietet verschiedenste Inputfelder, wie zum Beispiel Textinputfelder, Dateninputfelder Dropdownlisten und eine Vorlage für Farbauswahl. Für den Fall, dass es für den Anwendungsfall beispielsweise ein Auswahlfenster für die Font Awesome Icons (siehe Anhang B, Abbildung 18), keine Vorlage gibt, kann man dies einfach über den Telerik Animation Container selbst entwickeln.

Die Telerik UI-Komponenten sind vielfältig anpassbar, sodass man für normale Anforderungen einfach eine Oberfläche erstellen kann. Im Vergleich zu JavaScript-Bibliotheken, die dazu in Konkurrenz stehen, merkt man aber, dass die Einstellungen noch nicht weit genug gehen. Es ist beispielsweise nicht ohne Weiteres möglich, im Telerik Grid die Spalten über C#-Code zu tauschen. Die Untertabellen, der Event Typ Attribute, sind über ein Icon in der ersten Spalte standardmäßig auf- und einklappbar. Um diese Spalte zu verschieben, wurde hier übergangsweise ein Workaround verwendet, der mithilfe von CSS die Spalte in eine leere Spalte verschiebt.

### 4.4.3 Event Entries

Die Seite der Event Entries (siehe Anhang B, Abbildung 19) ist gleich aufgebaut wie die Seite der Event Typen. Auch hier ist ein Grid zusehen, in der die Event Entries tabellarisch dargestellt werden. Dieses Grid ist ebenfalls aufklappbar, sodass der Nutzer die Attribute sehen kann. Ebenfalls gibt es auch wieder verschiedene Modals, die über die Buttons im Grid oder über das `AppContentFrame` erreichbar sind. Im Gegensatz

zu den Attributen der Event Typen ist hier nur ein Modal verfügbar, um die Werte zu bearbeiten. Da die Attribute automatisch beim Erstellen der Event Entries miterstellt werden und es nicht vorgesehen ist, diese zu löschen, werden keine weiteren Modals benötigt. Da dieses Modal Werte von verschiedenen Datentypen (siehe Kapitel 4.2.5) haben kann, wird hier eine Komponente benötigt, die auf Basis des Datentyps das richtige Inputfeld anzeigt. Diese Komponente besitzt ebenfalls die Möglichkeit die Maßeinheiten der Attribute als Addon mit anzuzeigen, sofern eine Maßeinheit im Event Typ Attribut angegeben wurde.

Die Seite zeigt an mehreren Stellen sowohl in der Spalte Create das als auch in den Werten der Attribute des Datentyps DateTime, Daten und Uhrzeiten dar. Hier ergibt sich ein weiteres Problem: da Blazor Server auf dem Server ausgeführt wird, kennt Blazor hier nicht die Zeitzone des Endanwenders. Um dem Nutzer die Daten in der korrekten Zeitzone anzuzeigen, wird hier ein Service implementiert, der die Interoperabilität mit JavaScript nutzt. Über die `InvokeAsync` Methode des Interfaces `IJSRuntime` können hier JavaScript Methoden im Browser des Nutzers ausgeführt werden.

```
//TimeZoneService.js
function blazorGetTimezoneOffset() {
    return new Date().getTimezoneOffset();
}

//TimeZoneService.cs
public sealed class TimeZoneService
{
    private readonly IJSRuntime _js;
    private int _userOffsetInMinutes;

    public TimeZoneService(IJSRuntime js)
    {
        _js = js;
    }

    public async Task InitAsync()
    {
        _userOffsetInMinutes = await
      _js.InvokeAsync<int>("blazorGetTimezoneOffset");
    }

    public DateTime GetLocalDateTime(DateTime datetime)
    {
        return datetime.AddMinutes(-_userOffsetInMinutes);
    }

    public DateTime GetUtcDateTime(DateTime datetime)
    {
        return datetime.AddMinutes(_userOffsetInMinutes);
    }
}
```

Codeteil 11: TimeZoneService

Über diesen Service kann sichergestellt werden, dass eingegebene Daten und Uhrzeiten in die koordinierte Weltzeit umgewandelt werden können. So kann sichergestellt werden, dass in der Datenbank immer die Zeitzone UTC verwendet wird. Diese Daten und Uhrzeiten können dann anschließend wieder in die Zeitzone des Nutzers umgewandelt werden.

Über das `AppContentFrame` wird dieser Scoped-Service registriert (siehe Kapitel 4.4.1).

## 4.5 Buildprozess

Um die verschiedenen Anwendungen parallel zu entwickeln, wurden verschiedene Repositories verwendet (siehe Tabelle 6). Diese Projekte sind alle gleich aufgebaut und werden nach dem gleichen Muster benannt, wie man der Tabelle 6 entnehmen kann. Beispielsweise heißt das Repository der Web-Anwendung des elektronischen Logbuchs `SYSTECS.TestFactory.ELO.App`.

Jedes Repository besitzt zwei Submodule. Submodule erlauben es dem Entwickler ein Repository innerhalb eines Repositories zu benutzen. Dieses Projekt kann getrennt entwickelt werden und wird dann als Git-Repository in ein Unterverzeichnis eingebunden. Im Submodul der Build-Templates sind die verschiedenen Pipeline-Templates als YAML-Dateien gespeichert. Diese Templates werden für den Buildprozess der einzelnen Repositories benötigt. Das Submodul Shared enthält Dateien, die projektübergreifend genutzt werden. Beispielsweise ist hier die Konfiguration des ReSharpers enthalten, da alle Projekte nach den gleichen Einstellungen formatiert werden sollen.

Tabelle 6: Repositories der Test Factory

| **Repository** | **Name** |
|---|---|
| Project | SYSTECS.TestFactory.<Part>.<Component> |
| Build-Templates | SYSTECS.TestFactory.Build.Templates |
| Shared-Settings | SYSTECS.TestFactory.Shared |

Jedes Projekt enthält verschiedene CI/CD-Pipelines, um die Anwendung automatisch zu bauen und bereitzustellen:

- Die Pipeline **CI** fungiert als Überprüfung der Kontinuierlichen Integration. Diese Pipeline wird benutzt, um jeden Pull Request abzusichern. Die Pipeline baut das Projekt und überprüft den Code anhand der ReSharper Code Analyse und

bricht ab, sollte der Buildvorgang fehlerhaft sein oder die Überprüfung Fehler finden (siehe Kapitel 2.11).

- Die Pipeline **Sonar** überprüft den Code anhand der statischen Code-Analyse des SonarQubes. Die Ergebnisse der Analyse findet der Entwickler auf der Webseite des SonarQube-Servers oder mit der Visual Studio Erweiterung SonarLint in einem Fenster der Entwicklungsumgebung (siehe Kapitel 2.12).
- Die Pipeline **Service (develop)** erstellt ein pre-release NuGet Paket des Konnektors und komprimiert den Service, inklusive der REST-API, im ZIP-Dateiformat.
- Die Pipeline **Service (release)** erstellt ein NuGet Paket des Konnektors und komprimiert den Service, inklusive der REST-API, im ZIP-Dateiformat. Der Service wird anschließend über ein PowerShell-Skript auf Azure bereitgestellt.
- Die Pipeline **App (develop)** erstellt ein pre-release NuGet Paket der Webanwendung.
- Die Pipeline **App (release)** erstellt ein NuGet Paket der Webanwendung. Wenn es sich um die Hauptwebanwendung der Test Factory handelt, wird diese ebenfalls auf Azure bereitgestellt.

Bei den Pipelines handelt es sich um Azure Pipelines aus Azure DevOps (siehe Kapitel 2.1).

Die Pipelines können entweder mithilfe einer YAML-Datei oder der graphischen Oberfläche des Pipeline-Editors erstellt und bearbeitet werden. Der Editor hat den Vorteil, dass jeder Entwickler ohne großes Knowhow eine solche Pipeline definieren kann. Der große Vorteil der YAML-Dateien ist aber, dass diese Dateien wiederverwendbar sind. Die Dateien können ebenfalls in einem Repository eingecheckt und verwaltet werden. Um die Pipelines an das jeweilige Repository anzupassen, können Variablen verwendet werden. So kann man der Pipeline beispielsweise mitteilen, welches Projekt der Solution gebaut werden soll.

Das Verhalten der Versionierung kann mithilfe der `GitVersion.yml` definiert werden.

```yaml
next-version: 0.2.0
mode: ContinuousDelivery
branches:
    main:
        increment: Patch
        tag: ''
    release:
        increment: Patch
        tag: beta
    develop:
        increment: Patch
        tag: alpha
ignore:
  sha: []
merge-message-formats: {}
```

Codeteil 12: GitVersion.yml

Im Codeteil 12 erkennt man, dass in der Einstellung nach jeder Continuous Delivery die dritte Stelle der Version inkrementiert wird. Handelt es sich hier um den Release-Branch, wird der `beta` Tag angefügt und beim Develop-Branch wird der `alpha` Tag angefügt. Möchte der Entwickler die erste oder zweite Stelle der Version ändern, kann er in der Commit-Message des Pull Requests `+semver major` oder `+semver minor` hinzufügen. Dies überschreibt das Verhalten, das in der `GitVersion.yml` definiert wird.

Die einzelnen Projekte nutzen NuGet Pakete von verschiedenen Paketquellen. Diese Quellen sind in der `nuget.config` Datei hinterlegt. Da die Telerik-Pakete nur in einer privaten Quelle über einen Nutzeraccount verfügbar sind, wird jedes Paket, in den lokalen Test Factory NuGet Feed kopiert. Dies funktioniert mit dem Befehl `dotnet nuget push`.

## 4.6 Debugging und Fehlerbehandlung

Sollten trotz der verschiedenen Integrationstests und trotz der Codeanalyse Bugs oder Fehler auftreten hat der Entwickler mehrere Möglichkeiten das elektronische Logbuch zu debuggen:

- Wenn bekannte Fehler, die direkt von der REST-API behandelt werden können auftreten, gibt die REST-API eine Antwort mit dem dazu passenden HTTP-Statuscode zurück. Beispielsweise wenn versucht wird ein Event Type mit einem bereits bekannten Namen erstellen, antwortet die REST-API mit dem Statuscode 409 Conflict. Dieser Statuscode kann auch dem Nutzer der Webanwendung angezeigt werden.

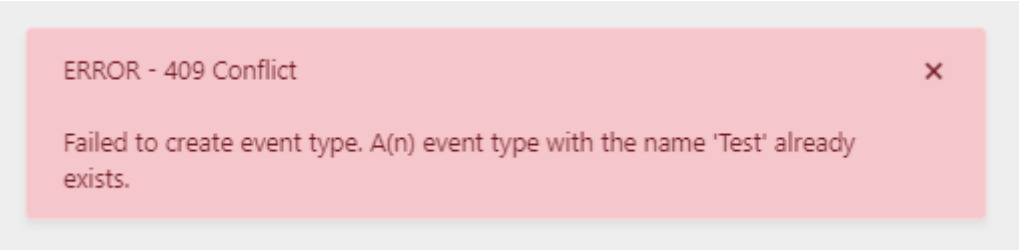


Abbildung 14: Konflikt beim Erstellen eines Event Types

- Bei jedem Request loggt die REST-API, welche Methoden aufgerufen wurden. Diese Daten sind in Azure auslesbar. So kann der Entwickler nachvollziehen, in welcher Methode der Fehler aufgetreten ist. Danach kann der Entwickler den Fehler auf seiner lokalen Umgebung rekonstruieren.
- Neben der Anwendung in Azure kann der Entwickler die Anwendung auch in einer lokalen Umgebung starten. Diese Umgebung wird zum Entwickeln und zur Fehlersuche genutzt.
- Manchmal ist auch eine Kombination der beiden Umgebungen nötig, wenn beispielsweise die Anwendung in Azure sich anders verhält als die lokale Anwendung. Der Codeteil 13 zeigt ein Beispiel, wie der Entwickler beim Ausführen des Webservices die lokale Datenbank oder die Datenbank in Azure ansteuern kann. Der gleiche Ansatz ist auch zwischen der Webanwendung und dem Webservice möglich. In der `launchSettings.json` wird die Variable `ASPNETCORE_ENVIRONMENT` festgelegt. Beim Starten der Anwendung kann dann eine überschriebene Version der `appsettings.json` genutzt werden.

```
//launchSettings.json
"Systecs.TestFactory.ELO.WebService - Development": {
  "commandName": "Project",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  },
  "applicationUrl": "https://localhost:44340/"
},
"Systecs.TestFactory.ELO.WebService - Local": {
  "commandName": "Project",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Local"
  },
  "applicationUrl": "https://localhost:44340/"
}

//appsettings.Local.json
"ConnectionStrings": {
    "ELODatabase": "Server=localhost;Database=ELO-
      DEV;Trusted_Connection=True;"
}

//appsettings.Development.json
"ConnectionStrings": {
    "ELODatabase": "CONNECTIONSTRING TO AZURE SQL DB"
}
```

Codeteil 13: LaunchSettings / AppSettings Konfiguration

# 5 Evaluation

Die zuvor vorgestellte Implementierung des elektronischen Logbuchs wird in diesem Kapitel anhand der Anforderungen aus Kapitel 3 evaluiert. Hierbei wird überprüft, ob diese Anforderungen in der aktuellen Implementierung erfüllt wurden.

## 5.1 Überprüfung der funktionalen Anforderungen

Die funktionalen Anforderungen wurden im Kapitel 3.1 anhand von Use-Case-Diagrammen beschrieben. Alle funktionalen Anforderungen, die die Webanwendung betreffen wurden, erfüllt. Der Nutzer kann über die Tabellen auf die Event Typen, Event Entries und deren Attribute zugreifen und diese mithilfe von Modals bearbeiten. Auch die Such- und Filterfunktion wurden implementiert. Die funktionalen Anforderungen, die die anderen Anwendungen der Test Factory betreffen, wurden ebenso erfüllt. Da die Webanwendung die gleiche REST-API nutzt, kann jede Applikation der Test Factory die gleichen Operationen durchführen. Da zu diesem Zeitpunkt aber noch keine andere Anwendung fertig implementiert wurde, die das elektronische Logbuch nutzen soll, konnten die Anforderungen noch nicht in der Praxis getestet werden. Gerade die Funktion der Properties `ObjectTypeName`, `ObjectTypeUrl`, `ObjectEntityName` und `ObjectEntityUrl` der Event Entries (siehe Kapitel 4.2.4) werden in einem solchen Praxistest in der Zukunft noch überprüft.

## 5.2 Überprüfung der nicht-funktionalen Anforderungen

Mit der Überprüfung der funktionalen Anforderungen konnte sichergestellt werden, dass das elektronische Logbuch funktioniert. Neben diesen Anforderungen sind die nicht-funktionalen Anforderungen für die Qualität einer Anwendung genauso notwendig. Die nicht-funktionalen Anforderungen wurden im Kapitel 3.2 beschrieben.

Die Anforderungen zur Wartbarkeit wurden erfüllt. Durch die Dokumentation der REST-API mit Swashbuckle kann jeder Entwickler schnell auf einen Blick alle Möglichkeiten, wie er mit dem elektronischen Logbuch interagieren kann, sehen.

Des Weiteren wurde im Kapitel 4.2.6 gezeigt, dass die Datenbank, auch bei vielen Einträgen, performant arbeitet. Da dies der kritische Teil für eine gute Performance der REST-API ist, wurde diese Anforderung auch erfüllt.

Im Kapitel 4.6 wurde gezeigt, dass die Webanwendung auch mit den zu erwartenden Ausnahmen umgehen kann. Problematisch ist hier aber, dass wenn ein Fehler auftreten sollte, der nicht behandelt werden kann, muss der Nutzer die Seite Anwendung neu laden. Der Benutzer bleibt danach auf der aktuellen Seite, muss aber gegebenenfalls das

Modal neu öffnen und die Daten erneut eingeben. Dies liegt an dem Framework ASP.NET Core Blazor Server, da in einem solchen Fall die SignalR-Verbindung unterbrochen wird. Somit wurde auch die Zuverlässigkeit zum großen Teil erfüllt.

Die Anforderungen zu der Benutzerfreundlichkeit konnten nur teilweise erfüllt werden. Da parallel zu dieser Arbeit eine andere Anwendung implementiert wurde, die auch in ihrer Webanwendung die Telerik Grid Komponenten nutzt, wurden einige Teile, in Bezug auf das Design und die Benutzerfreundlichkeit, doppelt implementiert. Diese Teile werden zurzeit extrahiert und können anschließend als eigene Komponente beiden und später auch allen anderen Anwendungen der Test Factory zur Verfügung gestellt werden. Aktuell ist dieser Schritt noch nicht abgeschlossen, deshalb sind die Anforderungen zur Benutzerfreundlichkeit zurzeit nicht erfüllt.

Da zurzeit dieser Implementierung noch keine Anwendung der Test Factory fertiggestellt ist kann sich hier in Zukunft noch einiges ändern.

## 5.3 Fehlende Anforderungen

Im Moment ist die REST-API noch nicht durch Authentifikation und Autorisierung abgesichert, da eine Authentifikation erst am Ende der Arbeit an die Test Factory hinzugefügt wurde. Dies kann man aber ohne großen Aufwand noch implementieren.

Wenn man ein NuGet-Paket in einer Blazor-Web-Applikation verwendet, müssen alle CSS- und JS-Dateien des Paketes ebenfalls von Hand eingebunden werden. Hier wäre eine automatische Lösung wünschenswert, sodass man keine Datei vergessen kann. Diese statischen Objekte können nicht alle automatisch importiert werden, da beispielsweise dann die Reihenfolge der CSS-Dateien nicht berücksichtigt werden kann. Auch hier wird noch an einer Lösung gearbeitet.

# 6 Zusammenfassung und Ausblick

Diese Bachelorarbeit hat sich mit der Konzipierung und Entwicklung des elektronischen Logbuchs der SYSTECS Test Factory beschäftigt. Hierbei wurde ein generisches Datenmodell erstellt, um die in Kapitel 3 genannten Anforderungen zu erfüllen. Danach wurde ein Webservice implementiert (siehe Kapitel 4.3), der eine REST-API enthält, sodass die später entwickelte Webapplikation (siehe Kapitel 4.4) und andere Anwendungen der Test Factory auf das Datenmodell zugreifen können.

Wie im vorherigen Kapitel gezeigt wurde, konnten die meisten Anforderungen an das elektronische Logbuch erfüllt werden. Auf Basis dieser Arbeit können die Nutzer an einem zentralen Ort die Status der Tests der verschiedenen Batteriezellen verfolgen.

Im nächsten Schritt kann die REST-API des elektronischen Logbuchs an eine andere Applikation der Test Factory angebunden werden. Hier soll das elektronische Logbuch in Verbindung mit einer anderen Anwendung getestet werden.

Danach können die restlichen Anforderungen noch implementiert werden. Ebenso wird das Design noch angepasst, wenn andere Anwendungen der Test Factory erstellt werden, sodass dieses Design wieder mit allen übereinstimmt.

# Anhang A: größere Codeteile

## Anhang A.1 Datentransferobjekt EventType

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Runtime.Serialization;
using JetBrains.Annotations;

namespace Systecs.ELO.DomainModel.ElectronicLogBook
{
    [Serializable]
    public sealed class EventType : ISerializable
    {
        private const int STRING_MAX_LENGTH_50 = 50;

        /// <summary>
        ///     Default constructor used for deserialization.
        /// </summary>
        public EventType()
        {
        }

        private EventType([NotNull] SerializationInfo info,
      StreamingContext context)
        {
            if (info == null)
            {
                throw new ArgumentNullException(nameof(info));
            }

            Id = info.GetGuid(nameof(Id));
            Icon = info.GetString(nameof(Icon));
            Color = info.GetString(nameof(Color));
            Status = info.GetBoolean(nameof(Status));
            Name = info.GetString(nameof(Name));
            EventTypeAttributes =
                info.GetValue<IList<EventTypeAttribute>>(nameof(
                EventTypeAttributes));
        }

        [Required] public Guid Id { get; set; }

        [Required(ErrorMessage = "Icon is required")]
        [StringLength(STRING_MAX_LENGTH_50,
          ErrorMessage = "Must be between 1 and 50 characters",
          MinimumLength = 1)]
        public string Icon { get; set; }

        [Required(ErrorMessage = "Color is required")]
        [RegularExpression(@"^#[0-9a-fA-F]{6}$", ErrorMessage =
          "Must have the following format: #XXXXXX")]
```

```
        public string Color { get; set; }
         [Required] public bool Status { get; set; }

         [Required(ErrorMessage = "Name is required")]
         [StringLength(STRING_MAX_LENGTH_50,
           ErrorMessage = "Must be between 1 and 50 characters",
           MinimumLength = 1)]
         public string Name { get; set; }

         public IList<EventTypeAttribute>
           EventTypeAttributes { get; set; }

         public void GetObjectData(SerializationInfo info,
           StreamingContext context)
         {
             if (info == null)
             {
                 throw new ArgumentNullException(nameof(info));
             }

             info.AddValue(nameof(Id), Id);
             info.AddValue(nameof(Icon), Icon);
             info.AddValue(nameof(Color), Color);
             info.AddValue(nameof(Status), Status);
             info.AddValue(nameof(Name), Name);
             info.AddValue(nameof(EventTypeAttributes),
                 EventTypeAttributes);
         }
    }
}
```

Codeteil 14: Datentransferobjekt EventType

## Anhang A.2 Generischer Zugriff auf die EventEntryAttributes

```
//EventEntryAttribute.cs
public async Task<EloActionResult<IList<EventEntryAttributeDto>>>
      GetEventEntryAttributesAsync(Guid eventEntryId)
{
    using var scope = new AutoMethodScopeLogger(_logger);

    var returnList = new List<EventEntryAttributeDto>();

    var instanceHelperDictionary =
          EventEntryAttributeTypeExtension
          .EventEntryAttributeInstanceHelperDictionary;
    foreach (var (valueInst, classInst) in
          instanceHelperDictionary)
    {
        returnList.AddRange(await
          GetEventEntryAttributesGenericAsync(
                classInst, valueInst, eventEntryId)
          .ConfigureAwait(false)
                as List<EventEntryAttributeDto> ?? new
                List<EventEntryAttributeDto>());
    }

    return Ok((IList<EventEntryAttributeDto>) returnList);
}

//EventEntryAttribute.Generic.cs
private async Task<List<EventEntryAttributeDto>>
      GetEventEntryAttributesGenericAsync<TClass, TType>(
      TClass _tClass, TType _tType, Guid eventEntryId)
      where TClass : class, IEventEntryAttribute<TType>
{
    var dbSet = DbContext.Set<TClass>();
    var entities = await
      dbSet.GetEventEntryAttributesAsync<TClass,
      TType>(eventEntryId).ConfigureAwait(false);

    return _mapper.Map<List<EventEntryAttributeDto>>(entities);
}

//EventEntryAttributeExtensions.cs
public static async Task<List<TClass>>
GetEventEntryAttributesAsync<TClass,
      TType>(this IQueryable<TClass> dbSet, Guid eventEntryId)
     where TClass : class, IEventEntryAttribute<TType>
{
    var entities = await dbSet
                        .Include(item => item.EventEntry)
                        .Include(item => item.EventTypeAttribute)
                        .ThenInclude(item => item.DataType)
                        .Where(item => item.EventEntryId ==
                            eventEntryId)
                        .ToListAsync()
                        .ConfigureAwait(false);

    return entities;
}
```

Codeteil 15: Generischer Zugriff auf die EventEntryAttribute

## Anhang A.3 Datenbank Benchmark

```
use [ELO-TEST]

DECLARE @EventTypesCount INT, @EventEntriesCount INT, @I INT, @J
INT
SET @EventTypesCount = 100
SET @EventEntriesCount = 10000
SET @I = 0
SET @J = 0

DECLARE @EventTypeId uniqueidentifier
DECLARE @EventEntryId uniqueidentifier

DECLARE @BoolId uniqueidentifier
DECLARE @StringId uniqueidentifier
DECLARE @DateTimeId uniqueidentifier
DECLARE @IntId uniqueidentifier
DECLARE @DoubleId uniqueidentifier

WHILE @I < @EventTypesCount
BEGIN
    SET @EventTypeId = NEWID()
    SET @BoolId = NEWID()
    SET @StringId = NEWID()
    SET @DateTimeId = NEWID()
    SET @IntId = NEWID()
    SET @DoubleId = NEWID()

    -- add Event Type
    INSERT INTO [dbo].[EloEventType]([Id], [Icon],
          [Color], [Status], [Name])
      VALUES(@EventTypeId, 'fa-bomb', '4CB050', 1,
        CAST(@EventTypeId AS VARCHAR(50)))

    -- add Event Type Attributes
    INSERT INTO [dbo].[EloEventTypeAttribute]([Id], [EventTypeId],
      [DataTypeId], [Name], [Dimension])
        VALUES(@IntId, @EventTypeId, 'FC377813-41A3-48EB-9892-
          284196993BF1', CAST(@IntId AS VARCHAR(50)), null)

    INSERT INTO [dbo].[EloEventTypeAttribute]([Id], [EventTypeId],
      [DataTypeId], [Name], [Dimension])
        VALUES(@StringId, @EventTypeId, 'CB32DAB7-71DB-4303-82E5-
          3AEA1DC5E840', CAST(@StringId AS VARCHAR(50)), null)

    INSERT INTO [dbo].[EloEventTypeAttribute]([Id], [EventTypeId],
      [DataTypeId], [Name], [Dimension])
        VALUES(@DateTimeId, @EventTypeId, '7307C067-4A47-4C9A-
          8724-48DCDA6A14DE', CAST(@DateTimeId
          AS VARCHAR(50)), null)

    INSERT INTO [dbo].[EloEventTypeAttribute]([Id], [EventTypeId],
      [DataTypeId], [Name], [Dimension])
        VALUES(@BoolId, @EventTypeId, '143C6187-F278-418A-AD58-
          6578E218554F', CAST(@BoolId AS VARCHAR(50)), null)
```

```
     INSERT INTO [dbo].[EloEventTypeAttribute]([Id], [EventTypeId],
           [DataTypeId], [Name], [Dimension])
             VALUES(@DoubleId, @EventTypeId,
                 '3ADC2172-94DB-4D3D-8FF6-72CAA1103B41',
                 CAST(@DoubleId AS VARCHAR(50)), null)

    WHILE @J < @EventEntriesCount
    BEGIN
        SET @EventEntryId = NEWID()
  -- add Event Entry
        INSERT INTO [dbo].[EloEventEntry]([Id], [EventTypeId],
      [CreatedDate], [UpdatedDate], [CreatedById], [UpdatedById])
             VALUES(@EventEntryId, @EventTypeId,
                 '2021-01-01', '2021-01-01', NEWID(), NEWID())

        SET @J = @J + 1
    END

    SELECT *
    INTO #TempEventEntry
    FROM dbo.EloEventEntry
    WHERE EventTypeId = @EventTypeId

    WHILE EXISTS(SELECT * from #TempEventEntry)
    BEGIN
        Select Top 1 @EventEntryId = Id From #TempEventEntry

        -- add Event Entry Attributes
        INSERT INTO [dbo].[EloEventEntryAttributeInt]([Id],
           [EventEntryId], [EventTypeAttributeId], [Value])
             VALUES(NEWID(), @EventEntryId, @IntId, 25)

        INSERT INTO [dbo].[EloEventEntryAttributeString]([Id],
      [EventEntryId], [EventTypeAttributeId], [Value])
             VALUES(NEWID(), @EventEntryId, @StringId,
                 'Sample String')

        INSERT INTO [dbo].[EloEventEntryAttributeDateTime]([Id],
      [EventEntryId], [EventTypeAttributeId], [Value])
             VALUES(NEWID(), @EventEntryId, @DateTimeId,
                 '2021-01-01')

        INSERT INTO [dbo].[EloEventEntryAttributeBool]([Id],
      [EventEntryId], [EventTypeAttributeId], [Value])
             VALUES(NEWID(), @EventEntryId, @BoolId, 1)

        INSERT INTO [dbo].[EloEventEntryAttributeDouble]([Id],
      [EventEntryId], [EventTypeAttributeId], [Value])
             VALUES(NEWID(), @EventEntryId, @DoubleId, 13.37)

        Delete #TempEventEntry Where Id = @EventEntryId
    END

    drop table #TempEventEntry

    SET @J = 0
    SET @I = @I + 1
END
```

Codeteil 16: SQL-Skript zum Einfügen von Testdaten

```
use [ELO-TEST]

DECLARE @EventEntryId uniqueidentifier
SET @EventEntryId = (Select top 1 Id from [ELO-
TEST].[dbo].[EloEventEntry] order by NEWID())

set statistics time on
set statistics io on


SELECT TOP(1) E.ID, E.EventTypeId, B.Value AS BoolValue,
     DA.Value AS DateTimeValue, DO.Value AS DoubleValue,
     I.Value AS IntValue, S.Value AS StringValue
FROM [ELO-TEST].[dbo].[EloEventEntry] AS E
INNER JOIN [ELO-TEST].[dbo].[EloEventEntryAttributeBool] AS B
ON E.Id = B.EventEntryId
INNER JOIN [ELO-TEST].[dbo].[EloEventEntryAttributeDateTime] AS DA
ON E.Id = DA.EventEntryId
INNER JOIN [ELO-TEST].[dbo].[EloEventEntryAttributeDouble] AS DO
ON E.Id = DO.EventEntryId
INNER JOIN [ELO-TEST].[dbo].[EloEventEntryAttributeInt] AS I
ON E.Id = I.EventEntryId
INNER JOIN [ELO-TEST].[dbo].[EloEventEntryAttributeString] AS S
ON E.Id = S.EventEntryId
WHERE E.Id = @EventEntryId

GO

set statistics time off
set statistics io off
```

Codeteil 17: Benchmark-SQL-Skript

# Anhang B: Abbildungen der Webseite

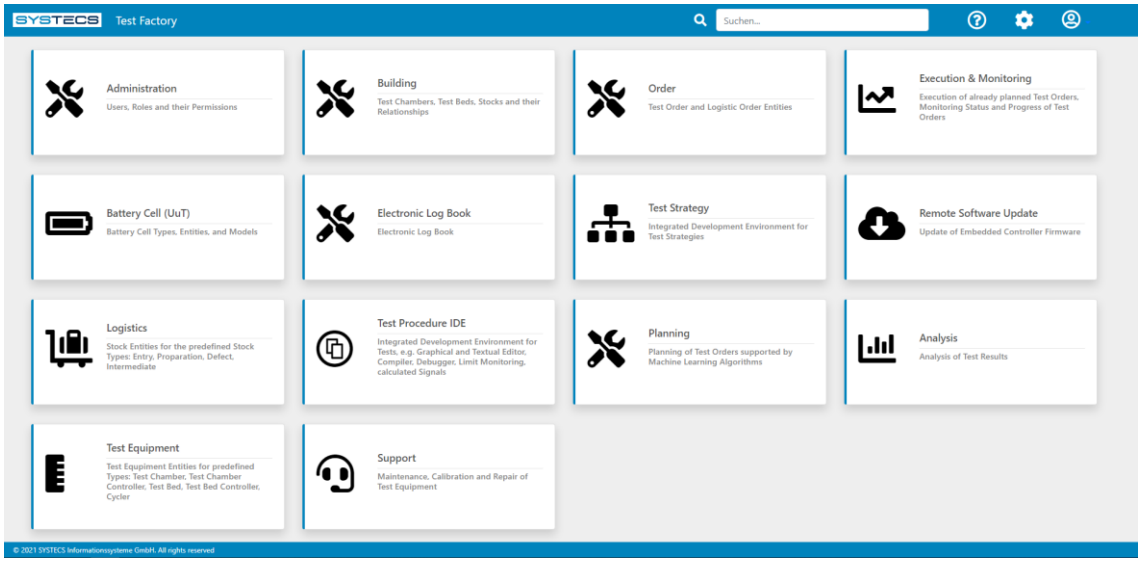


Abbildung 15: Webseite – Übersicht der Anwendungen

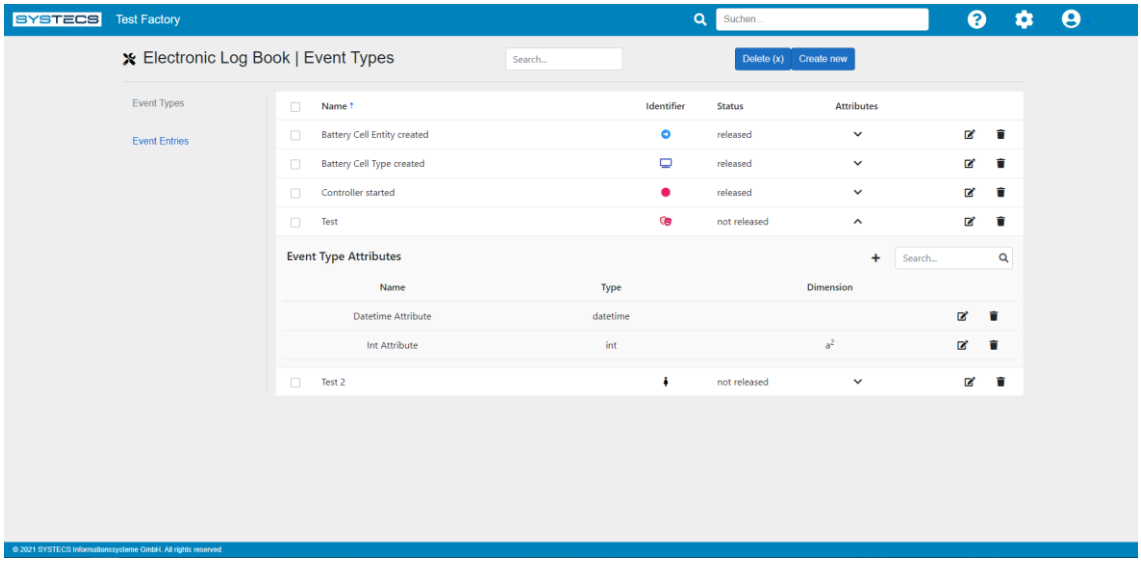


Abbildung 16: Webseite – ELO – Event Types

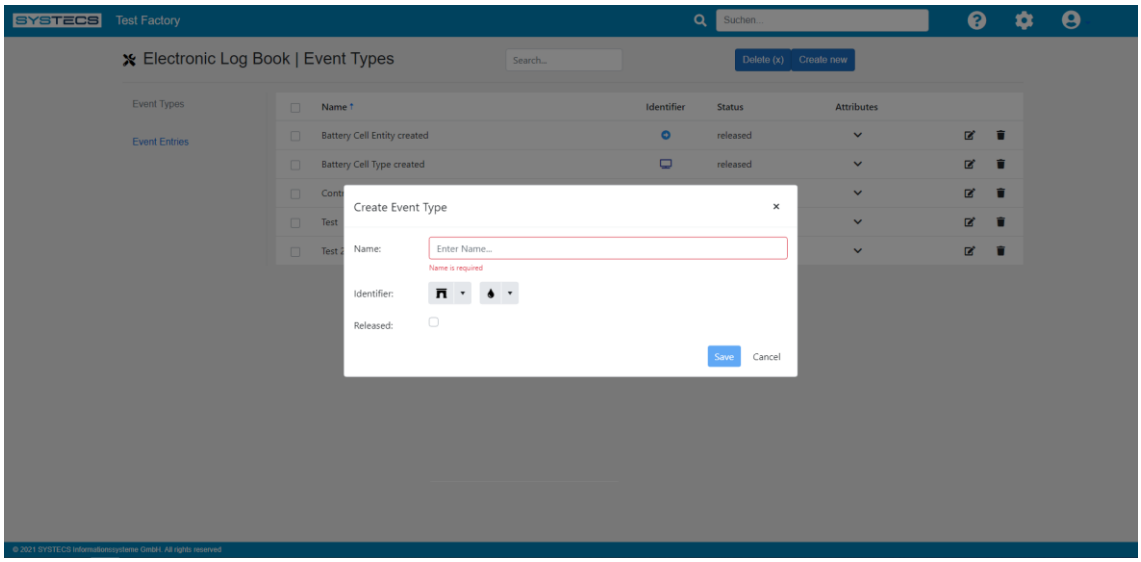


Abbildung 17: Webseite – ELO – Event Types – Create Modal

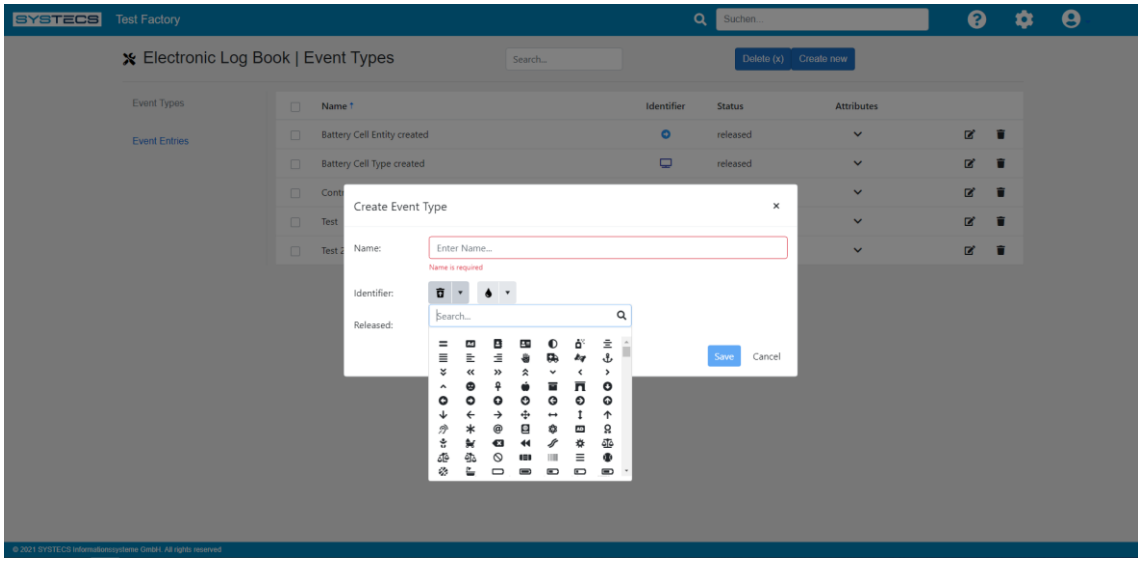


Abbildung 18: Webseite – ELO – Event Types – Create Modal – Icon Picker

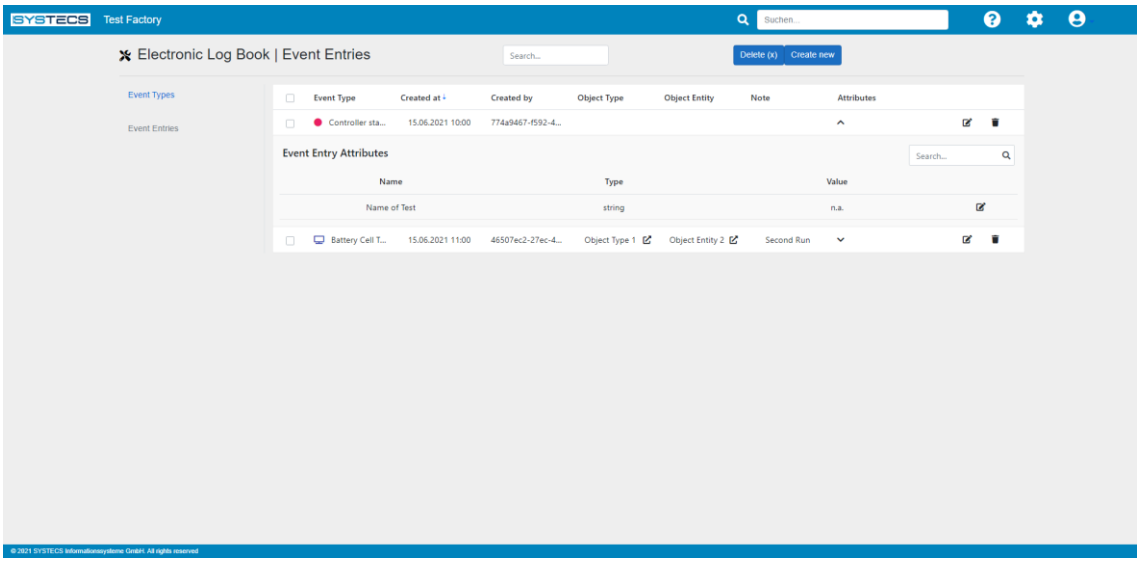


Abbildung 19: Webseite – ELO – Event Entries

# Literaturverzeichnis

**Bargaoanu, Lucian, et al. 2020.** AutoMapper. [Online] 25. juni 2020. [Zitat vom: 18. Juni 2021.] https://docs.automapper.org/en/stable/index.html.

**Douglas, Jon, et al. 2019.** An introduction to NuGet. [Online] Microsoft Corporation, 24. Mai 2019. [Zitat vom: 21. Juni 2021.] https://docs.microsoft.com/en-us/nuget/what-is-nuget.

**JetBrains s.r.o. o.D..** ReSharper - Die Visual-Studio-Erweiterung für .NET-Entwickler. [Online] JetBrains s.r.o., o.D. [Zitat vom: 21. Juni 2021.] https://www.jetbrains.com/resharper/.

**Latham, Luke, et al. 2020.** Introduction to ASP.NET Core Blazor. [Online] Microsoft Corporation, 25. September 2020. [Zitat vom: 6. Juni 2021.] https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0.

**Lin, Cephas, et al. 2020.** App Service overview. [Online] Microsoft Corporation, 6. July 2020. [Zitat vom: 17. Juni 2020.] https://docs.microsoft.com/en-us/azure/app-service/overview.

**Macanović, Mladen. 2021.** Blazorise. [Online] Megabit d.o.o., 5. Juli 2021. [Zitat vom: 12. Juli 2021.] https://blazorise.com/.

**Madole, Rob, et al. 2021.** Font Awesome. [Online] Fonticons, Inc., 16. März 2021. [Zitat vom: 21. Juni 2021.] https://fontawesome.com/.

**Microsoft Corporation. o.D..** Blazor. [Online] Microsoft Corporation, o.D. [Zitat vom: 18. Juni 2021.] https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor.

**—. o.D..** What is DevOps? [Online] Microsoft Corporation, o.D. [Zitat vom: 17. Juni 2021.] https://azure.microsoft.com/en-us/overview/what-is-devops/.

**Morris, Richard. 2021.** Swashbuckle.AspNetCore. [Online] 22. Mai 2021. [Zitat vom: 18. Juni 2021.] https://github.com/domaindrivendev/Swashbuckle.AspNetCore.

**Otto, Mark et al. 2021.** Build fast, responsive sites with Bootstrap. [Online] Twitter, Inc., 21. Mai 2021. [Zitat vom: 1. July 2021.] https://getbootstrap.com/.

**Ovhal, Pritam, et al. 2020.** What is Azure SQL? [Online] Microsoft Corporation, 27. July 2020. [Zitat vom: 17. Juni 2021.] https://docs.microsoft.com/en-us/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview.

**Progress Software Corporation. o.D..** Telerik UI for Blazor. [Online] Progress Software Corporation, o.D. [Zitat vom: 21. Juni 2021.] https://www.telerik.com/blazor-ui.

**Rossberg, Andreas. 2019.** WebAssembly Core Specification. [Online] World Wide Web Consortium, 5. Dezember 2019. [Zitat vom: 18. Juni 2021.] https://www.w3.org/TR/wasm-core-1/.

**SmartBear Software. 2020.** OpenAPI Specification. [Online] SmartBear Software, 20. Februar 2020. [Zitat vom: 18. Juni 2021.] https://swagger.io/specification/.

**SonarSource S.A. o.D..** SonarQube Documentation. [Online] SonarSource S.A, o.D. [Zitat vom: 21. Juni 2021.] https://docs.sonarqube.org/latest/.

**Synergy Research Group. 2021.** Cloud Market Ends 2020 on a High while Microsoft Continues to Gain Ground on Amazon. [Online] Synergy Research Group, 2. Februar 2021. [Zitat vom: 17. Juni 2021.] https://www.srgresearch.com/articles/cloud-market-ends-2020-high-while-microsoft-continues-gain-ground-amazon.

**Urban, Eric, et al. 2021.** What is Azure DevOps? [Online] Microsoft Corporation, 1. Januar 2021. [Zitat vom: 17. Juni 2021.] https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops.

**Vickers, Arthur, et al. 2020.** Entity Framework Core. [Online] Microsoft Corporation, 20. September 2020. [Zitat vom: 17. Juni 2021.] https://docs.microsoft.com/en-us/ef/core/.

**Warren, Matt. 2016.** Why is reflection slow? [Online] 14. Dezember 2016. [Zitat vom: 29. Juni 2021.] https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/.

**Weizenbaum, Natalie, et al. 2021.** Sass: Sass Basics. [Online] Google Inc., 23. Juni 2021. [Zitat vom: 2. Juli 2021.] https://sass-lang.com/guide.

**Wilson, Brad, et al. 2021.** About xUnit.net. [Online] .NET Foundation, 12. juni 2021. [Zitat vom: 18. Juni 2021.] https://xunit.net/.

**Zurawka, Thomas und Meyer, Olaf. 2019.** Software für eine vollautomatisierte, selbstlernende Prüffabrik für Batteriezellen. [Online] 25. März 2019. [Zitat vom: 20. Juli 2021.] https://www.systecs.com/fileadmin/user_upload/pdf/2019-03-25_software_f%C3%BCr_vollautomatisierte_selbstlernende_pr%C3%BCffabrik.pdf.

# Ehrenwörtliche Erklärung

| | | | |
|---|---|---|---|
| Name: | Meier | Vorname: | Max Manfred |
| Matrikel-Nr.: | 754468 | Studiengang: | SWB |

Hiermit versichere ich, Meier, Max, dass ich die vorliegende Bachelorarbeit mit dem Titel Entwicklung eines elektronischen Logbuchs für den Test von Batteriezellen selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebene Literatur und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Esslingen, den 28.07.2021 _____

Ort, Datum Unterschrift