Hochschule Esslingen
University of Applied Sciences

Entwicklung einer Webanwendung für den Test von Batteriezellen mit ASP.NET und Blazor

Bachelorarbeit

im Studiengang
Softwaretechnik und Medieninformatik

vorgelegt von

Mathias Herkelmann
Matr.-Nr.: 752437

am 15. Juli 2021
an der Hochschule Esslingen

Erstprüfer/in:	Prof. Dr.-Ing. Michael Scharf
Zweitprüfer/in:	Prof. Dr.-Ing. Harald Melcher

Kurzfassung

Die Herausforderungen der Elektromobilität sind eines der zentralen Themen der Industrie. Dabei besitzt jedes Elektrofahrzeug eine Batterie, welche wiederum aus Batteriezellen besteht. Diese besitzen komplexe Problemstellungen, welche gelöst werden müssen. Innerhalb dieser Bachelorthesis wird eine Webanwendung entwickelt, welche das Anlegen und Referenzieren dieser Zelltypen über eine Weboberfläche ermöglichen soll. Hierbei soll auch die Leistungsfähigkeit von ASP.NET und dem Blazor Server Framework überprüft werden.

Dieses Projekt ist hierbei nur ein kleiner Teil einer komplexen Prüffabrik, welche die Batteriezellen automatisiert testen soll, damit diese gefahrenfrei in Elektrofahrzeugen verwendet werden können.

Schlagwörter: Elektromobilität, Industrie, Batterie, Batteriezelle, Webanwendung, Framework, ASP.NET, Blazor.

Abstract

The challenges of electromobility are one of the central themes of industry. Each electric vehicle has a battery, which in turn consists of battery cells. These have complex problems that must be solved. Within this bachelor thesis, a web application is developed to enable the creation and referencing of these cell types via a web interface. The performance of ASP.NET and the Blazor Server Framework will also be checked.

This project is only a small part of a complex test factory, which is to test the battery cells automatically, so that these can be used safely in electronic vehicles.

Keywords: Electric mobility, industry, battery cell, webpage, framework, ASP.NET, Blazor.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

UI	User Interface
REST	Representational State Transfer
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
UI	User Interface
EF-Core	Entity Framework Core
.NET	ASP.NET Framework
SQL	Structured Query Language

1 Einleitung

Innerhalb dieser Bachelorthesis werden die relevanten Aspekte der Entwicklung einer Webanwendung mit ASP.NET und Blazor vorgestellt. Hierbei wird ein besonderes Augenmerk auf die von Microsoft zur Verfügung gestellten Technologien gelegt.

Zu Beginn wird erst einmal auf die Problemstellung eingegangen, welche dieses Projekt lösen will und anhand eines Rechenbeispiels dargestellt, wie dieses Problem zurzeit gelöst wird.

Im zweiten Kapitel werden die verwendeten Technologien sowie ihre Eigenschaften vorgestellt, welche im weiteren Verlauf dieser Arbeit wichtig sind.

Im dritten Kapitel wird ein besonderes Augenmerk auf die Konzeption der Webanwendung gelegt, sprich welche Anforderungen es zu erfüllen gilt. Auch wird hier die Konzeption der Datenbankstruktur anhand der Anforderungen vorgenommen.

Im vierten Kapitel wird die Realisierung der Webanwendung vorgestellt, welche Komponenten im Zuge dessen entwickelt wurden und warum diese wichtig sind, bzw. die Entscheidung für diese getroffen wurde.

Im fünften Kapitel wird die Implementierung anhand einiger Komponenten dargestellt. Hierbei wird dann auch auf die Funktionsweise eingegangen und wie diese Komponenten miteinander interagieren.

Im sechsten Kapitel soll das Projekt noch zusammengefasst und bewertet werden sowie ein Ausblick gewährt werden, welche Schritte noch erfüllt werden müssen.

1.1 Schilderung der Problemstellung

Die Elektromobilität ist immer weiter auf dem Vormarsch. Mittlerweile handelt es sich nicht mehr nur um Elektroautos, sondern auch andere Verkehrsmittel werden auf den Elektrobetrieb umgestellt. Der essentielle Bestandteil jedes Elektrofahrzeugs ist seine Batterie, welche wiederum aus verschiedenen Batteriezellen besteht (siehe Abbildung 1).

Abbildung 1: Elektrofahrzeug, Batterie, Batteriezelle

Um einen sicheren und effizienten Einsatz zu gewährleisten, müssen diese Batteriezellen umfangreich auf ihre Qualität und Leistungsfähigkeit geprüft werden. Hierfür werden intelligente Prüffabriken benötigt, welche diese Funktionalität jetzt und in Zukunft gewährleisten können.

Ziel ist hierbei, dass Verhalten der unterschiedlichen Batteriezellen zu untersuchen. Zum Einsatz kommen dabei sowohl statische als auch dynamische Kennlinien, welche zur Parametrisierung der Batteriemodelle verwendet werden.

Cell Type	**IMP 06160230P25A**
Description	Rechargeable **Li-ion** pouch cell
Cell Chemistry	Li-Ion
Supplier	Farasys Energy Inc.
Performance Specification	
Rated Capacity (min.)	-
Capacity (nominal)	25 Ah (e.g. **1C=25A**)
Capacity (typ)	-
Capacity (min)	23.5 Ah
Nominal Voltage	3.65 V
Operating Temperature	-20 °C to 60 °C
Gravimetric Energy Density	185 Wh/kg
Volumetric Energy Density	410 Wh/l
Charge Characteristic	
Charge Method	CC-CV
Charging Temperature	0 °C to 45 °C
max. Charge Current (cont.)	25 A
Standard Charge Current	-
Constant Voltage	4.15 V – 4.20 V
Charge Cutoff Current	250 mA
Charge Duration	-
Discharge Characteristic	
Discharge Method	CC
Discharging Temperature	-
max. Discharge Current (cont.)	100 A
Standard Discharge Current	-
Discharge Cutoff Voltage	2.0 V
Discharge peak (10s)	175 A
Storage Characteristic	
Storage Temperature	-
Physical Specification	
Cell Weight	485 g
Cell Width	161 mm
Cell Height (H2)	230 mm
Cell Height (H1)	240 mm
Cell Thickness (max)	6.0 mm

Abbildung 2: Datenblatt einer Batteriezelle

In Abbildung 2, werden die groben Eigenschaften einer Batteriezelle veranschaulicht, wie sie dem Datenblatt zu entnehmen sind, wie beispielsweise die Lade- und Entladecharakteristiken sowie die physischen Abmessungen. Wie zu sehen ist, sind dies viele unterschiedliche Parameter, welche hierbei zu beachten sind.

All diese Eigenschaften sind jedoch stark von der Temperatur, der zyklisch & kalendarischen Alterung sowie der Historie der Batteriezelle betroffen. Diese Informationen sind in den Datenblättern jeweils nur grob angegeben oder fehlen vollends. Dargestellt werden die Kennlinien in Abbildung 3.

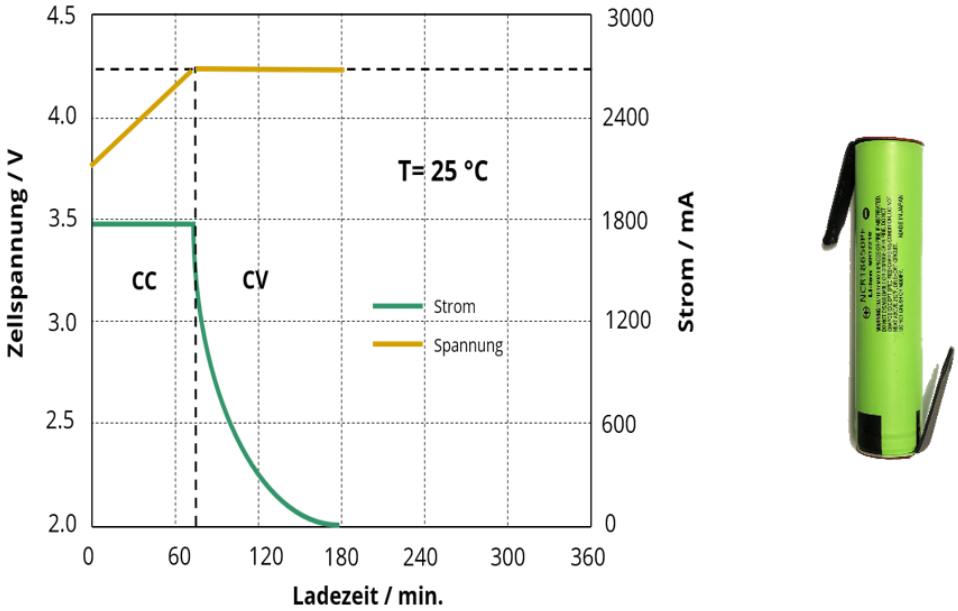


Abbildung 3: Kennlinien der Batteriezelle

Den Ladungsinhalt (State of Charge SoC) der Batterie während der Fahr zu erfassen ist eine der typischen Herausforderungen für den Fahrzeughersteller. Hierbei stellt sich allerdings das Problem, dass zwar bekannt ist, welche Ladung zu jeder Zeit entnommen wird, allerdings ist der aktuelle Ladungsinhalt durch die Einflüsse der Temperatur und Alterung der Batteriezellen nur schwer zu bestimmen. Je präziser diese Einflüsse bestimmt sind, umso besser kann der SoC festgestellt werden und hiermit auch die Ansteuerung der Batterie möglichst optimal erfolgen.

Der Markt für Elektrofahrzeuge wird in den nächsten Jahren voraussichtlich stark zunehmen. Das Hauptdifferenzierungsmerkmal der Hersteller ist hierbei die Leistungsfähigkeit der Batteriezelle bzw. die optimale Handhabung während der Fahrt (Speicherkapazität, Temperaturverhalten, Ladezeit, Alterung, etc.).

Um den effizienten Einsatz zu gewährleisten, müssen die Batteriezellen einer umfangreichen Prüfung innerhalb einer Prüffabrik unterzogen werden, um deren Verhalten genauer zu verstehen.

1.2 Stand der Technik

Die Prüfungen der Batteriezellen sind derzeit sehr kostenintensiv, da der Stromverbrauch, für viele Lade- sowie Entladezyklen enorm ist. Diese Prozesse müssen hierfür effizient

durchgeführt werden. Des Weiteren sind die Entwicklungszeiten für die einzelnen Prüfungen sehr hoch.

In einer Typischen Prüffabrik werden ca. 200 Prüfkammern, bestehend aus Klima- oder Temperaturschränken, mit jeweils 50 Prüfständen, eingesetzt. Jeweils eine Batteriezelle kann pro Prüfstand getestet werden, sprich es werden insgesamt 10.000 Batteriezellen gleichzeitig überprüft (siehe Abbildung 4).

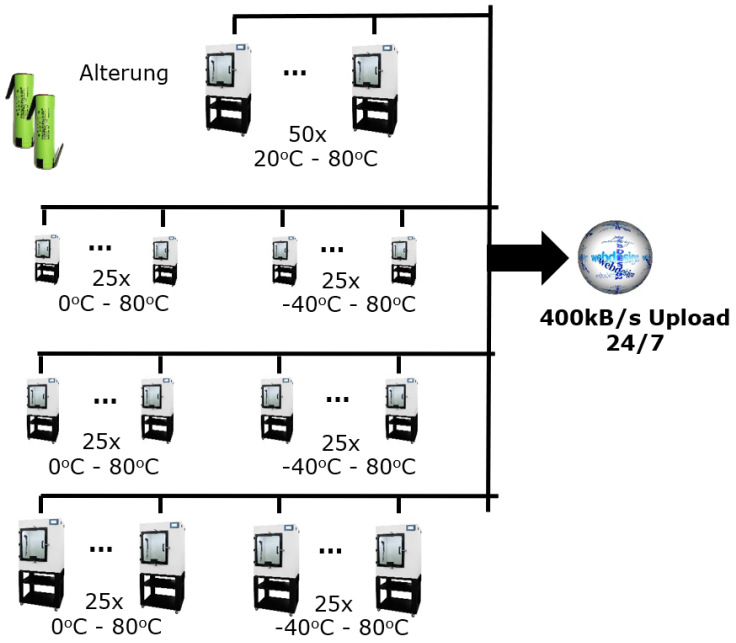


Abbildung 4: Aufbau einer typischen Prüffabrik

Eckdaten pro Prüfstand:

- Temperaturbereich: -40 °C – 80 °C
- Strombereich: -2A bis +2A
- Spannungsbereich: 0V bis 6V, -6V bis +6V
- Mittlere Prüfungsdauer: 8h
- Messdatenerfassung:
    - Abtastraten: 1s, 100 ms, 1 ms (nur für kurze Zeit)
    - Messung: Strom, Spannung, T Klimakammer, T UuT, sowie z.B. Leistung
    - Wortbreite: 16 Bit/32 Bit

Bei der folgenden Kalkulation werden im Mittel ein Strombedarf von 200 mA bei 8h/Tag Betrieb angesetzt. Diese Werte sind grob und können um eine 10-er Potenz abweichen (aber auch nicht mehr!).

D.h. pro Tag und Prüfstand:

- Energie: W = 6V * 200mA * 8h = **7,2 kWh**
- Messdatenmenge = 8h/100ms * 16 Bit = ca. **500 KByte**.

Eckdaten pro Prüfkammer mit 50 Prüfständen:

- Strom: 50 *200mA = 10A
- Energie: 50 * 7,2 kWh = 360 kWh pro Tag
- Messdatenmenge = 500 KByte * 50 = ca. **25 MByte**.

Eckdaten einer Prüffabrik mit 200 Prüfkammern:

- Energie: 200 * 360 kWh = **72 MWh pro Tag**
- Messdatenmenge = 25 MByte * 200 = ca. **5 GByte pro Tag**.

⇨ **Insbesondere der Energiebedarf ist beträchtlich und ein gewaltiger Kostenfaktor.**

2 Grundlagen

In den nachfolgenden Kapiteln werden die grundlegenden Technologien beschrieben, welche innerhalb der Erstellung dieser Bachelorthesis verwendet wurden. Hierbei soll zu Beginn eine kleine Einführung zur Entstehung und Weiterentwicklung von ASP.NET erfolgen. Darauffolgend werden die aus der ASP.NET Umgebung verwendeten Technologien noch einmal gesondert eingeführt. Danach werden weitere Technologien eingeführt, welche einen wichtigen Beitrag bei der Realisierung des Projekts geleistet haben.

2.1 Haupttechnologien

Im Folgenden werden die wichtigsten Technologien eingeführt, welche im Zuge dieser Arbeit verwendet werden. Das soll keinesfalls bedeuten, dass eine Technologie wichtiger ist als eine andere, sondern dies die zentralen Technologien sind.

2.1.1 ASP.NET

Das .NET-Framework (ausgesprochen dot-net) ist eine von Microsoft für das Windows-Betriebssystem entwickelte Laufzeitumgebung. Vom .NET-Framework wird eine Vielzahl von Diensten zur Verfügung gestellt. Es setzt sich aus der „Common Language Runtime“(CLR) und einer dazugehörigen Klassenbibliothek zusammen. (vgl. [1])

Die CLR wird hierbei für die Erstellung und Ausführung der Anwendungsprogramme benötigt. Die CLR kümmert sich hierbei unter anderem um die Typsicherheit, die Codegenauigkeit sowie um die Speicherverwaltung der Anwendung. (vgl. [1])

Die Klassenbibliothek stellt eine Vielzahl von Bibliotheken für die Softwareentwicklung bereit. Diese enthalten wiederverwendbaren Code, welcher bereits getestet wurde. Dieser kann problemlos in jede Anwendung eingebunden werden. Zudem werden auch die nötigen Programmierungsschnittstellen mitgeliefert. (vgl. [1])
Die folgenden Dienste werden vom .NET-Framework zur Verfügung gestellt:

- **Speicherverwaltung:** Bei vielen Programmiersprachen sind die Entwickler für die Zuordnung und Freigabe des Arbeitsspeichers sowie das Behandeln des Lebenszyklus der Objekte zuständig. In ASP.NET übernimmt die CLR diese Funktionalität und verwaltet das Speichermanagement für die angebunden Bibliotheken. (vgl. [1])
- **Typsystem:** Häufig werden auch die grundlegenden Typen einer Programmiersprache durch den Compiler definiert. Dadurch wird die sprachübergreifende Interoperabilität stark beschnitten. Durch das Typsystem von .NET ist

es möglich, die verschiedenen Dialekte, welche im .NET-Framework unterstützt werden, zu verwenden. (vgl. [1])

- **Klassenbibliothek:** Durch die Verwendung des .NET-Frameworks kann auf verschiedene Bibliotheken zugegriffen werden. Diese ermöglichen es allgemeine Programmiervorgänge auf einer niedrigen Ebene zu verwenden. (vgl. [1])
- **Entwicklungsframeworks und Technologien:** Die enthaltenen Bibliotheken halten Funktionalitäten für verschiedene Bereiche der Softwareentwicklung bereit. Zum Beispiel kann ASP.NET für Webanwendungen und ADO.NET für den Zugriff auf die Datenbanken verwendet werden. (vgl. [1])
- **Sprachinteroperabilität:** Der Sprachcompiler des .NET Framework wandelt den vom Entwickler geschrieben Code in die sogenannte CIL (Common Intermediate Language) um. Hierdurch können vom Entwickler seine bevorzugten Sprachen verwendet werden, da die Routinen der anderen Sprachen von .NET verfügbar gemacht werden. (vgl. [1])

Es gibt vier unterschiedliche Programmierungsmodelle, welche alle vom ASP.NET-Framework bereitgestellt werden. Diese nutzen allerdings unterschiedliche Entwicklungsansätze. Die Ansätze sind Web Forms, MVC, Web-Pages und Single-Pages. Im Kontext dieser Arbeit wird das Single-Page Model über Blazor Server verwendet, welches später innerhalb dieses Kapitels eingeführt wird.

Die Kommunikation zwischen dem Client und der ASP.NET- Webanwendung erfolgt über einen speziellen IIS-Webserver von Microsoft. Abhängig von der HTTP-Request, sucht der IIS-Webserver die entsprechende Applikation, welche von ihm gehostet wird und überprüft, ob der Aufruf autorisiert ist. Die Sicherheitsarchitektur von IIS wird in Abbildung 5 dargestellt.

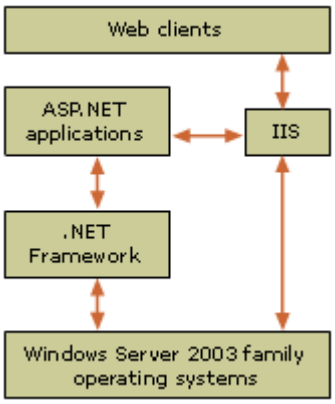


Abbildung 5: Aufbau der IIS-Sicherheitsarchitektur [2]

Die Webseiten werden vom Webserver instanziiert ausgeführt, für das Rendern an den Client übermittelt und nachfolgend gelöscht. Die Verbindung zwischen Server und Client ist hierbei statusfrei. (vgl. [2])

2.1.2 ASP.NET Core / ASP.NET 5

Seit 2014 gibt es parallel zu dem ASP.NET das ASP.NET Core-Framework. Es galt bis zur Version 3.1 als open-source Nachfolger von .NET. Die nachfolgende Version heißt nun .NET 5. Das Wegfallen der Bezeichnung „Core“ soll verdeutlichen, dass es sich hierbei ab jetzt um die Hauptimplementierung von .NET handelt. (vgl. [3])

Innerhalb dieser Arbeit wird ASP.NET 5 verwendet, allerdings wird die Bezeichnung „Core“ beispielsweise bei ASP.NET Core Blazor sowie Entity Framework Core weiterhin verwendet.

Der Hauptgrund für das neue Framework ist die modulare, cloud-optimierte, plattformübergreifende Architektur, welche als Open-Source bereitgestellt wird. Es ermöglicht die Erstellung von Webanwendungen, IoT-Diensten sowie mobilen Back-Ends. Auch werden die Entwicklertools nicht nur unter Windows, sondern auch für MacOS und Linux bereitgestellt. (vgl. [4])

Außerdem stehen verschiedene Hosting-Anbieter zu Verfügung, wie das schon unter ASP.NET bekannte IIS, aber auch neue Anbieter wie Kestrel, Nginx und Docker.

ASP.NET Core ermöglicht ebenfalls eine einheitliche Umgebung zum Erstellen der Webbenutzeroberfläche und von Web-APIs und unterstützt verschiedene Testing-Frameworks wie JUnit, welche für UnitTest verwendet werden können. (vgl. [4])
Diese Aufteilung in mehrere kleineren Bausteine, wird als NuGet-Packages bezeichnet.

2.1.3 ASP.NET Core Blazor

Bei Blazor handelt es sich um ein Framework zur Erstellung einer interaktiven clientseitigen Webbenutzeroberfläche mit .NET. (vgl. [5])

Sie umfasst das Erstellen der Benutzeroberfläche mit C# und der Razor Syntax statt dem ansonsten viel verwendeten JavaScript. Durch das gemeinsame Verwenden von Server- und Clientseitiger Logik fällt es sehr leicht, neue Webanwendungen zu erstellen. Gerendert wird die Benutzeroberfläche als HTML und CSS. Hierbei werden nahezu alle Browser (auch mobile) unterstützt. Auch können die bereits vorhandenen Bibliotheken und Funktionen des .NET- Ökosystems verwendet werden. (vgl. [5])

Der Name Blazor bildet sich aus den Begriffen Browser + Razor. Bei den sogenannten Razor-Pages handelt es sich um eine Technologie, welche das Programmieren von

seitenbasierenden Anwendungen erleichtert sowie die Produktivität erhöhen soll, im Vergleich zu den ansonsten verwendeten Controllern und Ansichten (vgl. [10]).

Es gibt verschiedene Ausführungsvarianten. Die zwei wichtigsten hierbei sind Blazor- Server und Blazor WebAssembly, welche in den nachfolgenden Kapiteln behandelt werden.

2.1.3.1 Blazor Server:

Bei Blazor Server wird die App innerhalb des ASP.NET Core Servers ausgeführt und die Aktualisierungen der Benutzeroberfläche, die Ereignisbehandlung sowie die Script-Aufrufe über eine SignalR-Verbindung geregelt (vgl. [7]). Diese SignalR-Verbindung zwischen Client und Server wird permanent und exklusiv zwischen dem jeweiligen Browser-Client und dem Server aufgebaut und ist persistent. Dies bedeutet, sollte ein Verbindungsabbruch auftreten, muss die Verbindung erneut aufgebaut werden, bevor erneut Datentransfer stattfinden kann. Die Struktur einer Blazor Server Awendung ist in Abbildung 6 dargestellt.

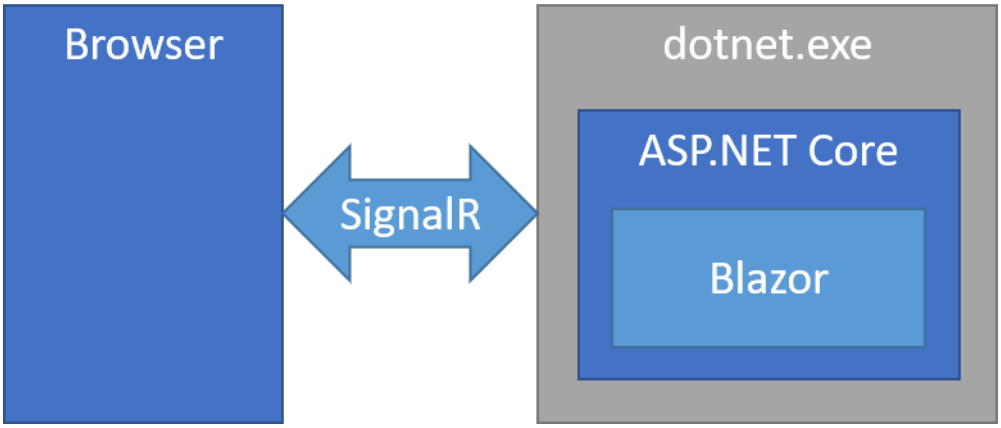


Abbildung 6: Struktur einer ASP.NET Core Blazor Server Anwendung [7]

Der Vorteil, welcher sich hieraus ergibt, ist, dass die Downloadmenge für den Client sehr gering ist, da nur die wichtigsten Darstellungseigenschaften heruntergeladen werden müssen. Hierdurch startet auch die Anwendung dementsprechend schneller als beispielsweise bei Blazor WebAssembly, welches im nächsten Kapitel besprochen wird. Dadurch kann Blazor Server auch gut mit weniger performanten Thin-Clients verwendet werden. Dadurch, dass die Logik des ausführbaren Quellcodes komplett auf dem Server verbleibt und nicht an den Client gesendet wird, ergeben sich Sicherheitsvorteile gegenüber anderen Webframeworks. (vgl. [7])

Hieraus resultieren allerdings auch Einschränkungen. Da jede Browserinstanz eine dauerhafte Verbindung zum Server aufbaut, ist die Teilnehmerzahl praktisch durch die Leistungsfähigkeit des Servers beschränkt (nicht skalierbar). Außerdem ist, da eine permanente Verbindung benötigt wird, kein Offlinesupport möglich. Zusätzlich

entsteht eine relativ hohe Netzwerklatenz, da z.B. jede Benutzereingabe erst an den Server gesendet werden muss. (vgl. [7])

2.1.3.2 Blazor WebAssembly:

Blazor WebAssembly baut auf der Technologie WebAssembly (kurz WASM) auf. Hierbei wird Bytecode zur Ausführung des nativen Codes verwendet.

Bei Blazor WebAssembly läuft der ausführbare Code innerhalb des Clients im Browser. Die hierfür nötigen Abhängigkeiten sowie die .NET Runtime, werden hierfür vom Browser heruntergeladen. (vgl. [7])

Die Anwendung wird hierbei innerhalb des User Interface (UI) des Browsers ausgeführt. Auch die Behandlung von Benutzereingaben sowie die Aktualisierung der Oberfläche werden hier behandelt. (vgl. [7])

Die Strukturierung im Browser wird durch Abbildung 7 verdeutlicht.

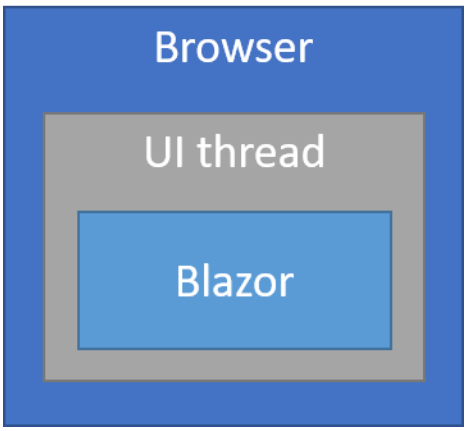


Abbildung 7: Aufbau der Blazor WebAssemlby-Anwendung [7]

Einer der Vorteile, welcher sich aus der Verwendung von WebAssembly ergibt, ist vor allem das schnelle Ausführen der Anfragen auf dem Client (vgl. [7]). Da der Code in einer Sandbox läuft und das DOM im Browser über JavaScript manipuliert, wird eine fast native Performanz erzielt. Allerdings kann in diesem Kontext nicht vollständig auf den Einsatz von JavaScript verzichtet werden.

Die Möglichkeiten des Clients sowie dessen Ressourcen, werden vollständig genutzt hierdurch kann man mittels des Clients beispielsweise auf die Kamera des Nutzers zugreifen. Außerdem kann WebAssembly von jedem Webserver, welcher Static File Deployment anbietet, wie u.a. CDN (Content Delivery Network), eingesetzt werden. (vgl. [7])

Es ist hierfür kein ASP.NET Core fähiger Server nötig, dadurch ist WebAssembly beliebig hochskalierbar. Auch ist im Gegensatz zu Blazor Server die Möglichkeit des Offline-Betriebs und die lokale Installation gegeben.

Der Nachteil, welcher sich ergibt, ist das der initiale Download zusätzlich die Assembly und WASM-Dateien beinhaltet und dadurch relativ groß ist. Dies führt zu einer verlängerten Startdauer der Anwendung. Außerdem werden nicht alle Webbrowser unterstützt und auch viele Funktionen der .NET Umgebung können nicht in WebAssembly genutzt werden. (vgl. [7])

2.1.4 Entity Framework Core

Das Entity Framework ist ein sogenannter ORM (Object Relational Mapper) hergestellt von Microsoft. Er wurde für die .NET-Plattform entwickelt und im Jahr 2008 zusammen mit dem .NET-Framework 3.5 veröffentlicht. (vgl. [8])

Bei Entity Framework Core erfolgt der Datenzugriff über ein Model. Dieses Model ist zusammengesetzt aus Entitätsklassen und dem Datenbankkontext. Der Kontext repräsentiert hierbei eine Session mit der Datenbank. Zusätzlich ist der Kontext auch für das Abfragen und Speichern der Daten innerhalb der Datenbank zuständig. (vgl. [8])

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

- **Codegenerierung** aus einer **existierenden Datenbank (Database first)** (vgl. [8]). Dieser Ansatz bietet sich an, wenn es bereits eine funktionstüchtige Datenbankanwendung oder es einen fähigen Datenbankentwickler gibt. Hierbei kann man mit Entity Framework die Entitätsklassen sowie den Datenbankkontext generieren lassen. Auch das Anpassen der Modelle nach Änderungen in der Datenbank sind hierdurch möglich, allerdings werden hierbei die alten Modelle überschrieben.
- **Datenbankgenerierung** aus **geschriebenem Code (Code first)** (vgl. [8]). Dieser Ansatz bietet sich an, wenn man sich die Datenbank generieren lassen möchte, da man beispielsweise bereits funktionierenden Code hat und nur auf eine neue Datenbank umsteigen möchte oder wenn der Entwickler kein Datenbankexperte ist. Hierbei werden zuerst die Entitätsklassen sowie der Datenbankkontext im Code beschrieben, ebenso wie die Relationen. Hieraus generiert EF-Core dann eine passende Datenstruktur in der Datenbank.

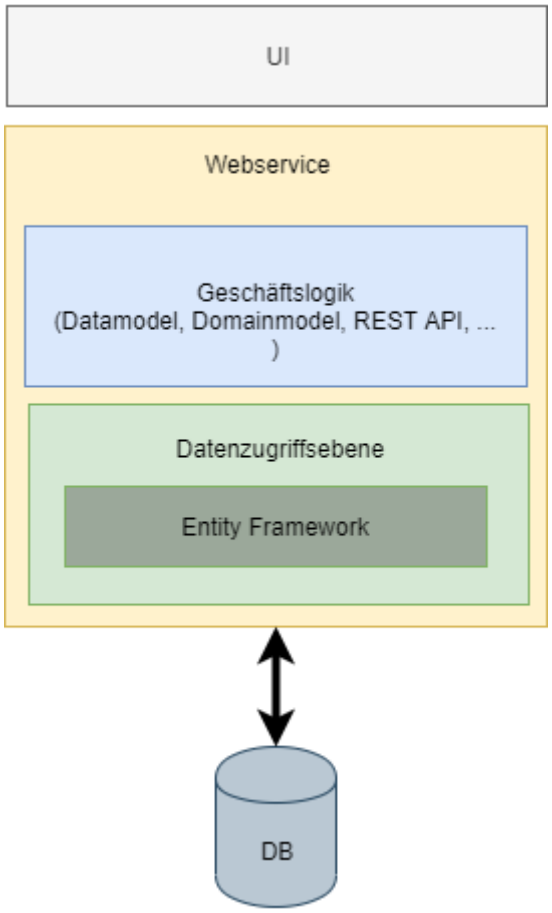


Abbildung 8: Darstellung der Ebenen von EF-Core

Entity Framework gliedert sich zwischen der Geschäftslogik und der Datenbank ein, wie in Abbildung 8 dargestellt. Es ist zuständig für das Speichern der Daten, welche in den Eigenschaften der Geschäftsentitäten vorhanden sind sowie für das Abrufen der Daten aus der Datenbank, welche es zu Entitätsobjekten umwandelt, welche dann im Code verwendet werden können.

2.2 Codeanalyse und Dokumentation

2.2.1 ReSharper

Der ReSharper ist eine heutzutage viel verwendete Erweiterung für Visual Studio (vgl. [9]). Der ReSharper bringt hierbei einige In-Build Funktionen mit. Die erste ist hierbei, wie der Name schon vermuten lässt das Re- Factoring von Code. Auch ist der ReSharper nützlich bei der Codebereinigung und Codeformatierung. Bereits zu Beginn des Entwicklungsprozesses kann hierbei ein Codestyle definiert werden, welcher dann bei allen Teammitgliedern einheitlich angewendet wird.

Zudem kann der Code nach Fehlern und Codequalitätsproblemen untersucht werden, hierbei werden Warnung und Empfehlungen schon beim Schreiben des Codes angeboten.

Auch bei der Codebearbeitung und Codegenerierung kann der ReSharper den Entwickler unterstützen, beispielweise können Interfaces automatisch generiert werden sowie fehlende „@using“ Anweisungen automatisch hinzugefügt werden.

Innerhalb dieses Projekts, wird der ReSharper sowohl beim Programmieren des Codes eingesetzt, wo er die Entwicklungszeit beschleunigt, da man nicht jeden Funktionsnamen selbst schreiben muss oder sich passende Interfaces direkt generieren kann.

2.2.2 SonarQube

Bei Sonar Qube handelt es sich um ein weiteres Tool zur Codeanalyse (vgl. [10]), welches allerdings im Bereich der Codeanalyse noch weitere Möglichkeiten als der ReSharper zur Verfügung stellt und sich damit zur Erweiterung desjenigen anbietet. Es erkennt Fehler sowie Sicherheitslücken als auch sogenannte Code-Smells, sprich schlecht geschriebenen Code.

Sonar Qube wird häufig zur Analyse des Codes bei der Continous Integration eingesetzt. Hierbei wird das Resultat der Analyse auf dem lokalen Server gespeichert. Für das Visual Studio gibt es eine Erweiterung namens SonarLint, welche es ermöglicht sich die Analyseergebnisse auch direkt im Visual Studio anzeigen zu lassen.

In diesem Projekt wird SonarQube eingesetzt, um die Git-Repositorys auf ihre Codequalität zu überprüfen.

2.2.3 Swashbuckle

Bei Swashbuckle [11] handelt es sich um eine OAS Implementierung für .NET.

OAS steht für OpenAPI Specification und in ihr wird die standardisierte, sprachunabhängige Spezifikation für die Beschreibung von RESTful APIs definiert (vgl. [12]).

Wie die Dokumentation über Swashbuckle aussieht, wird in Kapitel 4.2.4.3 dargestellt. Ebenso wird in Kapitel 5.1.5 beschrieben, wie man die Dokumentation über Swashbuckle vornimmt.

2.3 Weitere relevante Technologien

2.3.1 Azure DevOps

Bei DevOps handelt es sich um eine Komposition aus Development und Operations. Die zentrale Idee ist hierbei die Vereinigung der Prozesse, der Zusammenarbeit von Entwicklern und Technologien, um hochwertige Produkte zu erschaffen (vgl. [13]). Hierbei wirkt sich DevOps sowohl auf die Qualität der Software, die Verkürzung von

Entwicklungszeiten und die Testbarkeit aus. Zentraler Aspekt ist hierbei der DevOps-Anwendungslebenszyklus, welcher in Abbildung 9 dargestellt wird.

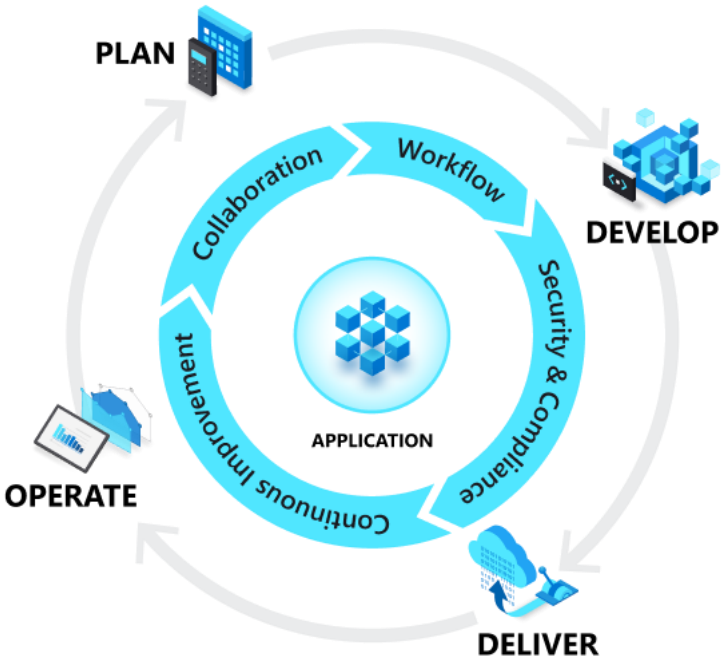


Abbildung 9: DevOps Anwendungslebenszyklus [13]

Dieser ist in vier Phasen unterteilt: Planung, Entwicklung, Bereitstellung und Betrieb

Hierbei wird der Entwickler bei der Zusammenarbeit, dem Arbeitsablauf, der Einhaltung der Sicherheitsstandards und der kontinuierlichen Verbesserung von DevOps unterstützt (vgl. [13]).

Hierbei werden mehrere Dienstleistungen von Azure DevOps zur Verfügung gestellt:

- Die **Azure Repos** ermöglichen den Einsatz von Git Repositories
- Die **Azure Pipelines** unterstützen Continous Integration (CI) und Continous Delivery (CD) Dienste für Build und Release.
- In den **Azure Plans** stehen verschiedene Tools für Applikationstests zur Verfügung.
- Agile Scrum- und Kanban-Tools werden von den **Azure Boards** verwaltet, welche zur Nachverfolgung von Fehlern und Problemen im Code genutzt werden können.
- Die **Azure Artifacts** kümmern sich um die Freigabe von Paketen (z.B. NPM-NuGet) aus privaten und öffentlichen Quellen sowie die Integration dieser in die Pipelines.
- Eine Dokumentation über die Anwendung kann im Wiki angelegt werden.

2.3.2 Azure

Die Cloudplattform Azure von Microsoft bietet über 200 Produkte an, wie unter anderem Webserver, Applikationsserver sowie Datenbanken. Hierbei können die Produkte individuell auf die Bedürfnisse der Benutzer angepasst werden, beispielsweise der Standort, die benötigte Datenmenge oder die Rechenleistung. (vgl. [14])

Innerhalb dieses Projekts werden neben der Azure SQL-Datenbank noch mehrere App Services beispielsweise für den Build- Prozess und das Deployment der Web-Applikation und Rest APIs verwendet.

2.3.2.1 Azure SQL Database

Bei der Azure SQL-Datenbank handelt es sich um eine relationale, skalierbare Datenbank, welche für die Cloudanwendung entwickelt und optimiert wurde. Sie basiert auf der Technologie SQL- Server von Microsoft und ist mit dieser quasi kompatibel. (vgl. [15])

Hierbei wird einem viel Arbeitsaufwand erspart, denn unter anderem kümmert sich Azure automatisch um das Aktualisieren der Datenbank sowie um Performance-Verbesserungen und Backups. (vgl. [15])

Zusätzlich werden einige Sicherheitsfeatures zur Verfügung gestellt, wie eine „advanced threat detection“ sowie einer eingebauten „Security-Control“, welche sich um T-SQL, Authentifizierung sowie Vernetzung und das Key-Management kümmert. (vgl. [15])

Zum anderen ist die Datenbank über das Azure Portal nach den Wünschen des Anwenders konfigurierbar.

2.3.2.2 Azure App Service

Der Azure App Service bietet sich zum Hosten von Webapplikation, REST APIs sowie mobilen Backends an (vgl. [16]).

Hierbei werden verschiedenste Sprachen, wie die Hauseigenen .NET und .NET Core, aber auch Node.js, PHP, Python, Ruby oder Java unterstützt.

2.3.3 NuGet

NuGet ist der Paketmanager für .NET (vgl. [17]). Ein NuGet- Paket ist zusammengesetzt aus bereits kompiliertem Code und anderen Dateien, welche innerhalb des Projekts genutzt werden. Zusätzlich ist auch eine Paketbeschreibung mit einer Versionsnummer enthalten. Hierbei handelt es sich um ZIP- Dateien, welche sowohl auf privaten als auch auf öffentlichen Hosts zur Verfügung gestellt werden können.

Die NuGet- Pakete können dann im Visual Studio zu Projekten in der gewünschten Version hinzugefügt werden

2.3.4 Xunit

Bei Xunit handelt es sich um ein Werkzeug, mit welchem automatisierte Unittests ausgeführt werden können. Es eignet sich vor allem zur Verwendung in Kombination mit .NET. (vgl. [18])

Hierdurch kann die Integrität des Codes gewährleistet werden. Zusätzlich kann man mittels Code Coverage sicherstellen, welcher Anteil des Codes durch Tests abgedeckt wird.

2.3.5 Moq

Bei Moq handelt es sich um eine Mocking- Bibliothek für das .NET- Framework. Es ermöglicht das Simulieren und Verifizieren von Interaktionen mit Abhängigkeiten, um Unittests zu ermöglichen bzw. zu erleichtern. (vgl. [19])

Eine Erläuterung, welche Vorteile Mockingframeworks liefern, wird in Kapitel 4.2.4.4 gegeben.

2.3.6 Automapper

Automapper ist ein simples Tool, welches als NuGet- Paket in Projekte eingebunden werden kann. Es handelt sich hierbei um einen Objekt- Mapper, welcher für das Mapping von einem Eingabeobjekt zu einem Ausgabeobjekt verwendet wird (vgl. [20]).

Die Funktionsweise wird in Kapitel 4.2.4.1 erklärt und in Kapitel 5.1.4 ein Beispiel anhand von Code dargestellt.

2.3.7 Telerik Blazor UI

Zusätzlich wird innerhalb dieser Arbeit die kostenpflichtige Telerik Blazor UI verwendet. Dies ermöglicht die Darstellung von verschiedenen vorgefertigten Komponenten. Insgesamt bietet Telerik über 75 vorgefertigte UI- Komponenten für Blazor an. Die angebotene Palette ist weitreichend und bietet unter anderem Buttons, Charts, Modals, Grids, Navigationbars sowie Forms und Dropdownmenüs.

Diese Komponenten sind native Blazor Komponenten und können sowohl unter Blazor Server als auch unter Blazor WebAssembly verwendet werden. Eingebunden werden sie als NuGet- Paket.

Es gibt verschiedene Themes, wie das standardmäßig von Telerik angebotene sowie ein Bootstrap- und ein Materials- Theme. Da in diesem Projekt zusätzlich auch Bootstrap für die Darstellung verwendet wird, wurde sich auch beim Theme für den Bootstrapansatz entschieden.

2.3.8 Font Awesome

Font Awesome ist eine Bibliothek, welche als NuGet- Paket in das Projekt eingebunden werden kann. Es enthält über 7800 verschiedene Icons, welche auf der Webseite angezeigt werden können (vgl. [21]). Hierbei werden verschiedene Styles für die jeweiligen Icons angeboten, wie unter anderem: Solid, Regular, Light, Duotone und Brands.

Die Icons können ganz einfach, wie im folgenden Codebeispiel eingebunden werden:

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

Codeteil 1: Font Awesome Beispiel

Hierbei handelt es sich um das User Symbol. Durch die Bezeichnung `fas` wird angegeben, dass hierbei der Style Solid verwendet werden soll.

3 Konzeption

In diesem Kapitel soll die Konzeption der BatteryCell- Webanwendung beschrieben werden, bevor in den folgenden Kapiteln die Umsetzung dargestellt wird. Hierbei liegt der Fokus auf der Formulierung von Anwendungsfällen sowie den funktionalen- und nichtfunktionalen Anforderungen der BatteryCellApp.

3.1 Spezifikation

3.1.1 Kurzbeschreibung des Systems

Es soll eine Webanwendung zum Test von Batteriezellen entwickelt werden. Es soll möglich sein, die Batterietypen mit ihren zugehörigen Abhängigkeiten anlegen zu können, wie u.a. ihrer Physischen Spezifikation, Performance Spezifikation, Ladecharakteristiken etc. Des Weiteren können Batterieentitäten angelegt werden, welche diese Zelltypen verwenden und zusätzliche Informationen zum Zelltypen enthalten. Später soll es dann möglich sein, diese Zelltypen zu verwenden, um die Testläufe anzulegen.

3.1.2 Anwendungsfälle

Im Folgenden werden die Anwendungsfälle bzw. Anforderungen an das System aufgelistet.

1. Der Nutzer soll einen neuen Zelltyp anlegen können. Hierbei hat er die Möglichkeit, direkt alle Abhängigen Werte einzutragen (Lade- und Entladecharakteristiken, Leistung etc.), oder nur die generellen Informationen des Zelltyps anzulegen und später alle weiteren Daten einzutragen.
2. Der Nutzer soll in der Lage sein, die angelegten Zelltypen im Nachhinein noch bearbeiten zu können oder Zelltypen zu entfernen.
3. Auch soll es möglich sein, gleich mehrere Zelltypen oder Zellentitäten zu entfernen.
4. Der Nutzer soll eine neue Zellentität anlegen können, welche Informationen wie das Lieferdatum, Seriennummer und Auftragsnummer enthält. Auch kann hier die Abhängigkeit der Zelltypen angelegt werden.
5. Die Entität kann auch im Nachhinein noch editiert oder gelöscht werden.

3.1.3 Balsamiq Prototyp

In diesem Projekt wird ein Balsamiq Prototyp verwendet, welcher als Vorlage für den Aufbau und Inhalt der Webanwendung verwendet wird. Er gibt bereits vor, welche Komponenten für die Anwendung entwickelt werden sollen. Ausschnitte des Prototyps werden

im Anhang B1 und B2 dargestellt. Aus diesem Prototyp konnten auch einige der Anforderungen entnommen werden, wie z.B. welche Daten auf der Übersichtsseite dargestellt werden sollen.

3.2 Funktionale Anforderungen

Der Nutzer muss …

- über das User Interface eine Zusammenfassung der Informationen aller Zelltypen bekommen.
- über das User Interface eine Zusammenfassung der Informationen aller Zellentitäten bekommen.
- über das User Interface die Möglichkeiten haben, die Darstellung der Daten zu durchsuchen und zu filtern.
- die CRUD- Operationen über ein Modal ausführen können.
- In der Lage sein, nur die generellen Informationen anzulegen und den Rest der Werte als Default zu initialisieren.

Jede Anwendung der Test Factory muss …

- Über die Schnittstellen der REST- API auf die Funktionen der BatteryCellApp zugreifen können.

Sowohl Nutzer als auch jede Anwendung der Test Factory muss …

- neue Zelltypen erstellen können
- neue Zellentitäten erstellen können.
- bestehende Zelltypen editieren können.
- Bestehende Zellentitäten editieren können
- In der Lage sein, Zelltypen, welche keiner Zellentität zugeordnet sind, zu entfernen.
- In der Lage sein, Zellentitäten zu entfernen.
- Mehrere Zelltypen auswählen und entfernen können.
- Mehrere Zellentitäten auswählen und entfernen können.

3.3 Nicht- funktionale Anforderungen

Das User Interface muss …

- an das Design der anderen Anwendungen der Test Factory angepasst sein.
- gute Bedienbarkeit aufweisen.
- gut strukturiert sein.
- gut aussehen

Die BatteryCellApp muss …

- einfach erweiterbar sein.
- zuverlässig sein.

3.3.1 Datenmodell

Aus den formulierten Anwendungsfällen kann das Datenmodell abgeleitet werden, wie in Anhang A1 dargestellt.

Die Entitäten PhysicalSpecification, PerformanceSpecification, ChargeCharacteristic, DischargeCharacteristic und StorageCharacteristic hängen vom jeweilen Zelltypen ab und enthalten verschiedene Informationen zur Batteriezelle, welche dem Datenblatt entnommen werden können. Sie sind alle mit einer 1:1 Beziehung verbunden. Dies bedeutet, jedem Zelltyp sind seine Spezifikationen eindeutig zugeordnet und wenn ein neuer Zelltyp angelegt wird, werden seine dazugehörigen Spezifikationen erstellt.

Die Zellentität ist mittels einer 1: n Beziehung mit dem Zelltyp verbunden, sprich jede Zellentität besitzt nur einen Zelltyp, der Zelltyp kann aber beliebig oft in Zellentitäten verwendet werden.

3.3.1.1 CellEntity

Die Relation `CellEntity` beinhaltet die Definition einer Zellentität und die Referenz auf den zugeordneten Zelltyp.

Tabelle 1: Datenbanktabelle CellEntity

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	CellEntity-ID
CustomId	nvarchar (max)		Id für den Nutzer
SerialNumber	nvarchar (max)		Seriennummer
DeliveryDate	Datetime2(7)		Datum der Lieferung
OrderNumber	nvarchar (max)		Ordernummer
Location	nvarchar (max)		Ort der Zelle / Verfügbarkeit
Comment	nvarchar (max)		Kommentar
CellTypeId	uniqueidentifier	FK	CellType-ID
TimeStamp	Datetime2(7)		Zeitstempel der letzten Änderung

3.3.1.2 CellType

Die Relation `CellType` enthält die Referenz auf alle ihre Kind- Elemente sowie die Definition des Zelltyps. Die Daten innerhalb des Zelltyps müssen angelegt werden, das Anlegen seiner Kind- Elemente ist optional. Sollten keine Werte für die Kind- Elemente vergeben werden, werden sie mit Defaultwerten initialisiert und können zu einem späteren Zeitpunkt definiert werden.

Tabelle 2: Datenbanktabelle CellType

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	CellType-ID
SupplierId	uniqueidentifier	FK	Supplier-ID
PhysicalSpecificationId	uniqueidentifier	FK	PhysicalSpecification-ID
PerformanceSpecificationId	uniqueidentifier	FK	PerformanceSpecification-ID
ChargeCharacteristicId	uniqueidentifier	FK	ChargeCharacteristic-ID
DischargeCharacteristicId	uniqueidentifier	FK	DischargeCharacteristic-ID
StorageCharacteristic	uniqueidentifier	FK	StorageCharacteristic-ID
Name	nvarchar (max)		Name des Zelltyps
Description	nvarchar (max)		Beschreibung des Zelltyps
Chemistry	Bit		Chemische Zusammensetzung der Zelle
Flag	Bit		
TimeStamp	Datetime2(7)		Zeitstempel der letzten Änderung

3.3.1.3 PhysicalSpecification

Tabelle 3: Datenbanktabelle PhysicalSpecification

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	PhysicalSpecification-ID
Construction	Tinyint		Um welche Zellart handelt es sich (z.B. Pouch Cell)
Weight	float		Gewicht der Zelle
WeightUnit	tinyint		Gewichtseinheit der Zelle
Height	float		Höhe der Zelle
HeightUnit	tinyint		Einheit der Höhe

3.3.1.4 PerformanceSpecification

Tabelle 4: Datenbanktabelle PerformanceSpecification

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	PerformanceSpecification-ID
RatedCapacity	Float		Geschätzes Fassungsvermögen
RatedCapacityUnit	Tinyint		Einheit des Fassungvermögens
CapacityNominal	Float		Nominal Kapazität der Zelle
CapacityNominal-Unit	Tinyint		Einheit der Kapazität
CapacityType	Float		Kapazitätstyp der Zelle
CapacityTypeUnit	Tinyint		Einheit des Kapazitätstyp
CapacityMin	Float		Minimal Kapazität

CapacityMinUnit	Tinyint		Einheit der minimalen Kapazität
NominalVoltage	Float		Nennspannung der Zelle
NominalVoltageUnit	Tinyint		Einheit der Nennspannung
OperatingTemperatureMin	Float		Minimale Betriebstemperatur
OperatingTemperatureMinUnit	Tinyint		Einheit der minimalen Betriebstemperatur
OperatingTemperatureMax	Float		Maximale Betriebstemperatur
OperatingTemperatureMaxUnit	Tinyint		Einheit der maximalen Betriebstemperatur
GravimetricEnergyDensity	Float		Gravimetrische Energiedichte
GravimetricEnergyDensityUnit	Tinyint		Einheit der gravimetrischen Energiedichte
VolumetricEnergyDensity	Float		Volumetrische Energiedichte
VolumetricEnergyDensityUnit	Tinyint		Einheit der volumetrischen Energiedichte

3.3.1.5 ChargeCharacteristic

Tabelle 5: Datenbanktabelle ChargeCharacteristic

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	ChargeCharacteristic-ID
Method	Tinyint		Lademethodik
TemperatureMin	Float		Min. Temperatur
TemperatureMinUnit	Tinyint		Min. Temperatur Einheit

TemperatureMax	Float		Max. Temperatur
TemperatureMaxUnit	Tinyint		Max. Temperatur Einheit
MaxCurrent	Float		Max. Stromfluss
MaxCurrentUnit	Tinyint		Max. Stromfluss Einheit
StandartCurrent	Float		Normaler Stromfluss
StandartCurrentUnit	Tinyint		Normaler Stromfluss Einheit
ConstantVoltage	Float		Konst. Spannung
ConstantVoltageUnit	Tinyint		Konst. Spannung Einheit
CutoffCurrent	Float		Reststrom
CutoffCurrentUnit	Tinyint		Reststrom Einheit
Duration	Int		Dauer
DurationUnit	Tinyint		Dauer Einheit

3.3.1.6 DischargeCharacteristic

Tabelle 6: Datenbanktabelle DischargeCharacteristic

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	DischargeCharacteristic-ID
Method	Tinyint		Entlademethodik
TemperatureMin	Float		Min. Temperatur
TemperatureMinUnit	Tinyint		Min. Temperatur Einheit
TemperatureMax	Float		Max. Temperatur
TemperatureMaxUnit	Tinyint		Max. Temperatur Einheit
MaxCurrent	Float		Max. Stromfluss
MaxCurrentUnit	Tinyint		Max. Stromfluss Einheit
StandartCurrent	Float		Normaler Stromfluss

StandartCurrentUnit	Tinyint		Normaler Stromfluss Einheit
CutoffVoltage	Float		Abschalte Spannung
CutoffVoltageUnit	Tinyint		Abschalte Spannung Einheit
PeakVoltage	Float		Spitzenspannung
PeakVoltageUnit	Tinyint		Spitzenspannung Einheit

3.3.1.7 StorageCharacteristic

Tabelle 7: Datenbanktabelle StorageCharacteristic

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	StorageCharacteristic-ID
TemperatureMin	Float		Min. Temperatur
TemperatureMin-Unit	Tinyint		Min. Temperatur Einheit
TemperatureMax	Float		Max. Temperatur
TemperatureMa-xUnit	Tinyint		Max. Temperatur Einheit

3.3.1.8 Supplier

Tabelle 8: Datenbanktabelle Supplier

Spaltenname	**Datentyp**	**Key, NULL**	**Kommentar**
Id	uniqueidentifier	PK	Supplier-ID
Name	Nvarchar (max)		Name des Zelltyps
Address	Nvarchar (100)		Beschreibung des Zelltyps
City	Nvarchar (100)		Name der Stadt
PostalCode	Nvarchar (16)		Postleitzahl
Country	Nvarchar (100)		Name des Lands

E-Mail	Nvarchar (max)		E-Mail-Adresse des Suppliers
Phone	Nvarchar (max)		Telefonnummer des Suppliers

3.4 Architektur

Das Projekt ist in 3 Schichten unterteilt. Hierbei besteht die Datenschicht aus einer Azure SQL- Datenbank, die Anwendungsschicht aus dem Webservice und die Präsentationsschicht aus der App (siehe Abbildung 10).

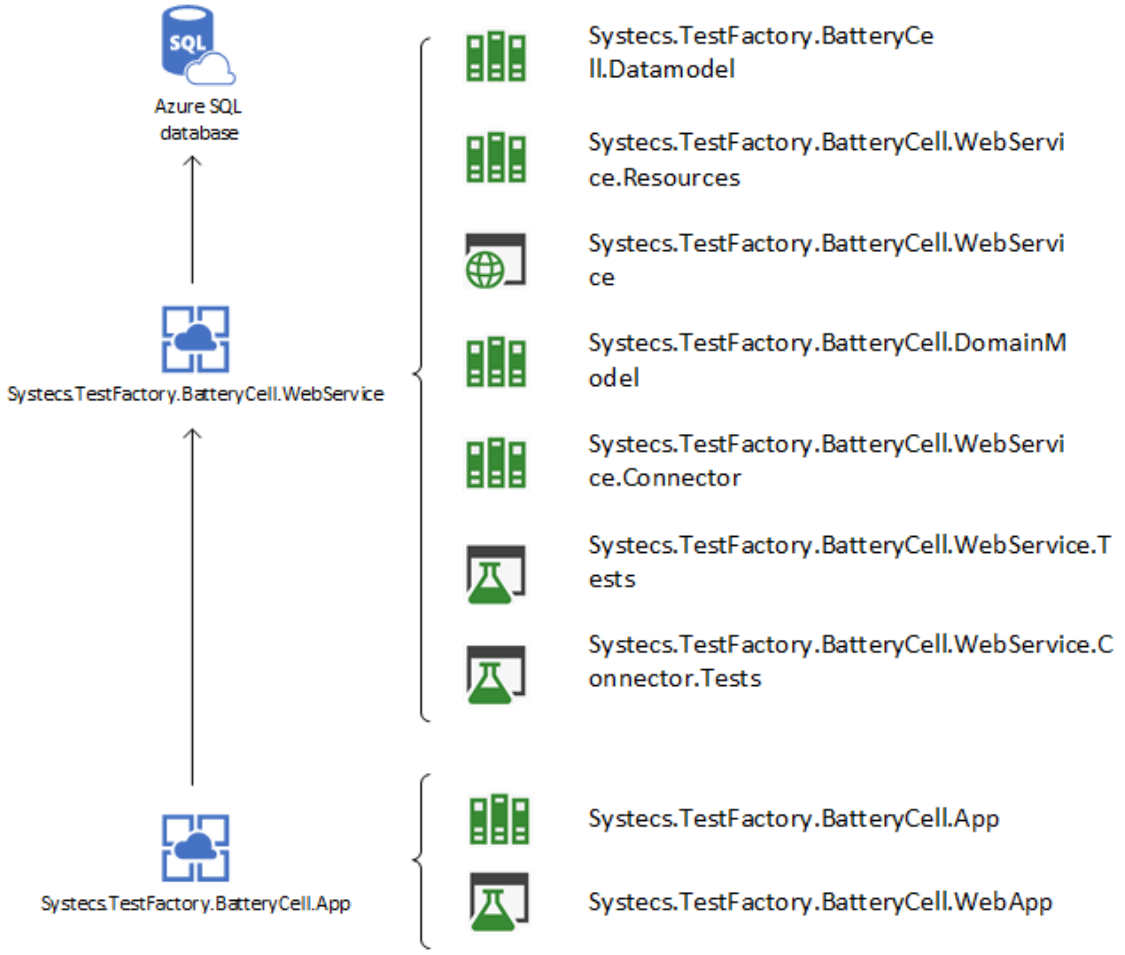


Abbildung 10: Architektur der Webanwendung

Auf die Umsetzung sowie die Funktion der einzelnen Komponenten wird in Kapitel 4 und 5 eingegangen.

4 Realisierung

In diesem Kapitel wird darauf eingegangen, in welcher Reihenfolge die Komponenten entwickelt wurden, warum dies so vorgenommen wurde und welche Erfahrungen dabei gemacht wurden. Ausgelassen wird hierbei die eigentliche Implementierung der Komponenten. Die wichtigsten Komponenten und wie dieses interagieren, ist im Kapitel Implementierung beschrieben.

4.1 Datenmodell

Begonnen wurde mit der Datenschicht. Um eine passende Datenbankstruktur zu erstellen, muss zuerst das Datenmodell erstellt werden. Die Vorlage für die Entitäten und die passenden Attribute bildeten die in den Datenblättern enthaltenen Information, diese wurden dann in dazu passende Datenbankentitäten zerlegt, um sie in der Datenbank logisch getrennt ablegen zu können. Beispielsweise welche physischen Abmessungen besitzt die Batteriezelle, welche Ladecharakteristiken weist sie auf usw. All diese Eigenschaften werden dann über ihre Beziehung zum Zelltyp wieder logisch gebündelt. Hierbei wurde eine Normalisierung bis zur 3.ten Normalform vorgenommen. Anhand des Anforderungskatalogs konnten die passenden Beziehungen zwischen den Entitäten angelegt werden. Später mussten die Datentypen einiger Entitäten noch überarbeitet werden, da die Telerik Blazor UI für die Checkbox nur Boolean als Datentyp akzeptiert. Des Weiteren enthielt die Zellentität zunächst noch eine Produktionszeit, welche sich später als unwichtig herausgestellt hat. Zu einem Späteren Zeitpunkt soll noch eine Veränderung für die PhysicalSpecification folgen, da hier die unterschiedlichen Zelltypen nicht korrekt abgebildet werden können, da beispielsweise eine Pouch- Zelle andere Abmessungen aufweist als eine Zylinderförmige- Zelle.

4.2 Anwendungsschicht

4.2.1 Datamodel

Auf Vorlage des Datenmodells konnten dann die passenden Entitätsklassen in der Anwendungsschicht angelegt werden. Hierfür wird Entity Framework Core verwendet. Gewählt wurde hierbei der Ansatz „Code First“, sprich es werden zunächst die Entitätsklassen erstellt, welche die Entitäten widerspiegeln, die von Entity Framework später in der Datenbank erstellt werden. Hierbei konnten auch die passenden Relationen zwischen den Entitäten bereits angelegt werden. Gewählt wurde der „Code First“ Ansatz, da er die Entwicklungszeit der Datenbank beschleunigt. Des Weiteren wurde eine solche Funktionalität in der Hochschule nicht behandelt, so dass dies eine neue Erfahrung darstellte.

4.2.2 Domainmodel

Nachdem das Datamodel vollständig angelegt wurde, konnte man sich mit der Struktur der Daten innerhalb der Software beschäftigen. Hierfür wurde der Prototyp der Anwendung verwendet, welcher darstellt, wie die Daten auf dem Frontend ausgegeben werden sollen. Da die Anwendung eine andere Struktur der Daten verwendet, als diese aufgrund der Normalisierung in der Datenbank abgelegt sind, wird auf das Domain- Driven- Design zurückgegriffen. Hierbei handelt es sich um einen Ansatz, bei dem Modelle erstellt werden, welche die Fachlichkeit der Anwendung widerspiegeln. Hierbei wird das Verhalten des Objekts, welches sich im Arbeitsspeicher befindet, implementiert, sprich das Objekt, welches dann in der Datenbank abgelegt werden soll (vgl. [21]).

Verwendet wird dies, da beispielsweise in der Datenstruktur eine klare Trennung von CellType und Supplier vorgenommen wird, allerdings werden diese in der Anwendung verwendet, als würden sie sich auf einem Level miteinander befinden. Hierfür erstellt man ein Domainmodell, welches diese Beziehung abbildet und kann diese Domainmodels dann über Tools wie den Automapper in das passende Datenmodell umwandeln. Auch die Umwandlung von Datenmodell zu Domänenmodell ist hiermit möglich. Das Domänenmodell wird als Aufbau der Objekte innerhalb der Anwendung genutzt, so wird immer erst ein Domänenmodell erstellt und dieses für das Übertragen der Daten in die Datenbank umgewandelt.

4.2.3 Ressources

Daraufhin wurde die Ressource erstellt. Diese definieren Fehlertexte, welche ausgegeben werden sollen, damit man diese nicht mehrfach schreiben muss. Diese können dann beliebig verwendet werden. Da zu Beginn noch nicht alle Fehlerfälle definiert waren, wurde die Ressource, während des Projekts immer wieder erweitert oder auch Fehlerfälle entfernt, nachdem ihr Auftreten gelöst wurde.

4.2.4 Webservice

Nachdem die Modelle so weit bestanden, konnte mit der Entwicklung des WebService begonnen werden, welcher die Geschäftslogik sowie die REST API- Schnittstellen enthält.

4.2.4.1 Mapping

Zuerst wurde allerdings hier das Mapping zwischen den Domain- und Datamodels implementiert. Hierfür wurde die Technologie Automapper verwendet. Hierdurch kann das Modell in beide Richtungen umgewandelt werden. Dafür muss nur das passende NuGet-Paket eingebunden werden.

4.2.4.2 REST- API

Nachdem das Mapping geregelt wurde, konnte der Controller für die REST API angelegt werden. Dieser verwendet verschiedene Dienste, um den Datenzugriff auf die Datenbank mittels Entity Framework zu regeln. Das Anlegen dieser Services war recht zeitaufwendig, da hierbei sehr darauf geachtet werden muss, welche Arten von Fehlern hierbei auftreten können, wie beispielsweise, dass der Entitätsname, welcher neu angelegt werden soll bereits existiert, oder dass die Anfrage nicht vollständig ist.

4.2.4.3 Dokumentation der Schnittstellen

Diese Schnittstellen müssen hierfür auch dokumentiert werden. Dafür wurde die Technologie Swashbuckle verwendet. Hiermit kann die Dokumentation der API einer Anwendung automatisch erstellt werden. Die Implementierung hierfür ist in Kapitel 5.1.5 beschrieben.

Hieraus entsteht dann eine Zusammenfassung des Endpoints, in welchem die einzelnen Parameter sowie die möglichen Rückgabetypen beschrieben werden (siehe Abbildung 11 und 12).

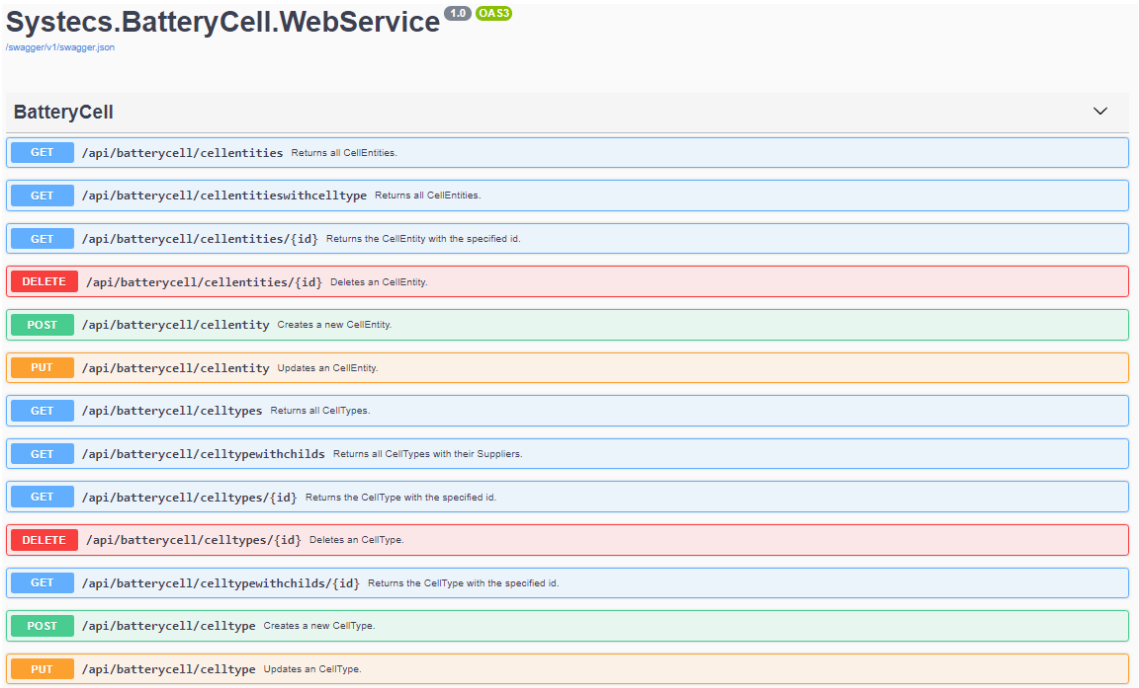


Abbildung 11: Ausschnitt Swashbuckle Dokumentation

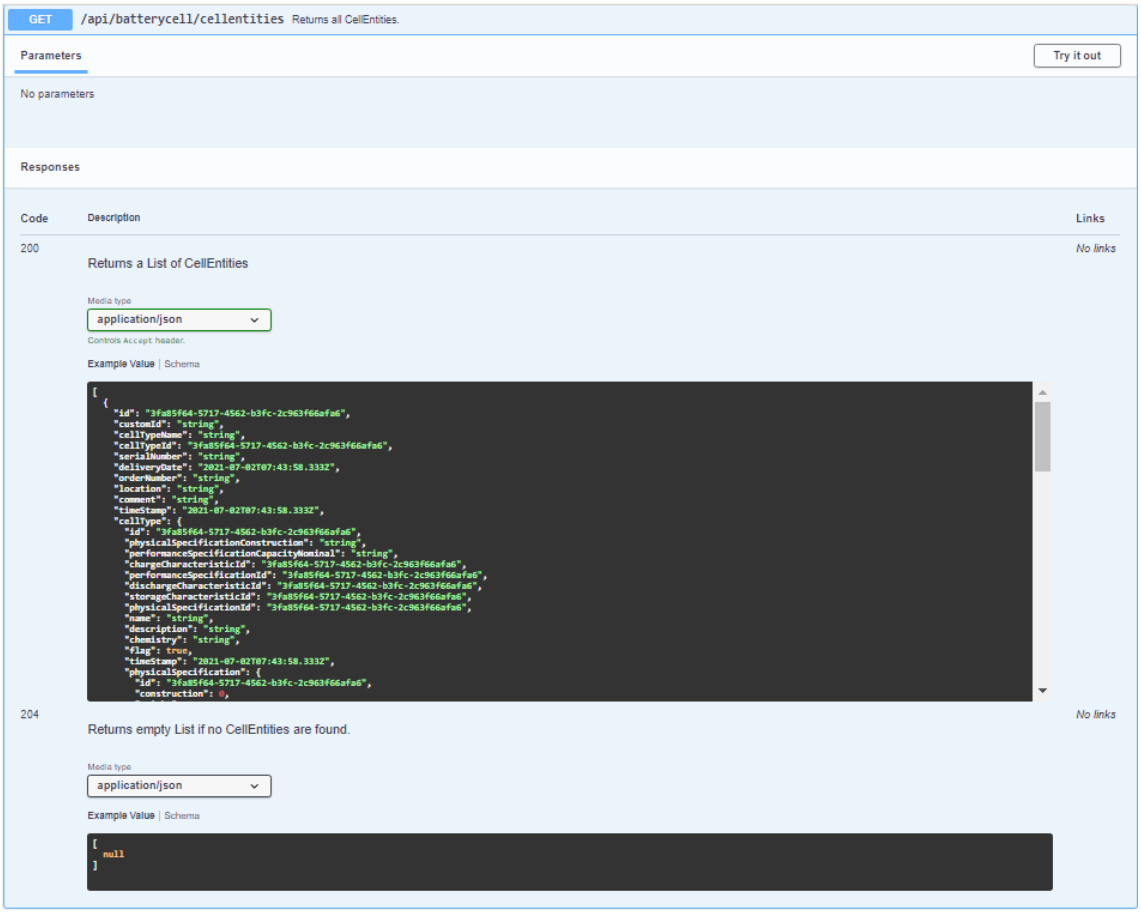


Abbildung 12: Swashbuckle Dokumentation des Endpoints /api/batterycell/cellentities

4.2.4.4 Integrationstests für den Webservice

Um sicherstellen zu können, dass jede Schnittstelle des Webservices erwartungsgemäß Daten zurückgibt, wurden dafür eigens Integrationstests geschrieben. Diese sollen die korrekte Funktionsweise der Schnittstellen gewährleisten und den Entwickler direkt informieren, ob durch seine Änderungen gegebenenfalls die Methoden nicht mehr erwartungsgemäß funktionieren und welche Schnittstellen betroffen sind.

Die Integrationstests wurden mit dem Tool xUnit und der Mocking Software Moq durchgeführt. Über Mockingsoftware können Platzhalter für nicht realisierte Objekte erstellt werden, dies ermöglicht es schon frühzeitig, Modultests durchzuführen (vgl. [23]).

Die Funktionsweise von Mocks wird in Abbildung 13 veranschaulicht.

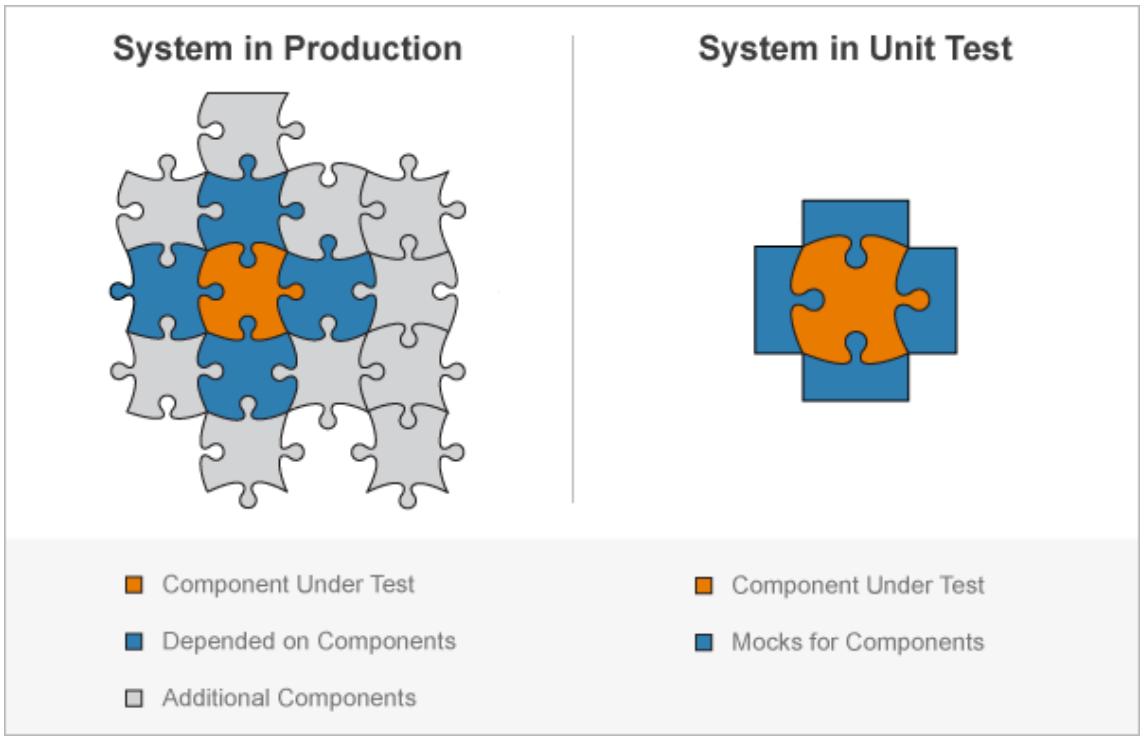


Abbildung 13: Mocking Software [24]

In diesem Fall ermöglicht es das Testen der Endpoints, ohne die Initialisierung des Webservices und dessen Komponenten ausführen zu müssen. Hierbei reicht es, die Umgebung mit einem Mock zu simulieren und die vom Endpoint gelieferten Rückgabewerte zu überprüfen.

4.2.4.5 Connector

Nachdem die Schnittstellen dokumentiert und getestet wurden, schloss sich die Entwicklung des Connectors an. Der Connector enthält die Methoden, um auf die Schnittstellen der REST API zuzugreifen. Er wird zusammen mit den Domainmodels als NuGet- Paket zusammengefasst und kann dann von beliebigen Anwendungen für das Zugreifen auf die REST API verwendet werden. Der Vorteil, der hieraus resultiert, ist das sich diese Anwendungen dann nicht um etwaige Änderungen in den Schnittstellen kümmern müssen, da hierfür nur der Connector aktualisiert werden muss. Außerdem entfällt das Einbinden der Logik des Zugriffs im Frontend.

4.2.4.6 Unit- Tests für den Connector

Um die Funktionalität des Connectors ebenfalls sicherzustellen, wurden für seine Methoden eigens Unit Tests erstellt, welche jede Methode gegen Testdaten in der Datenbank testen. Für die Tests wurde das Tool xUnit verwendet. Jede Methode des Connectors greift auf eine der REST Schnittstellen zu, indem sie diese entweder per GET abfragt oder

Daten mittels POST/PUT sendet. Die aus den übergebenen Daten erwarteten Rückgabewerte werden infolgedessen überprüft. Hierbei wird auch sichergestellt, dass korrekt auf Fehlerfälle reagiert wird.

Nachdem durch mehr als 50 Tests die Funktionalität der Methoden des Controllers gewährleistet werden konnten, wurde mit der Entwicklung der Weboberfläche begonnen. Ziel hierbei ist, dass Backend voll funktionstüchtig zu haben, wobei alle Funktionen bereits implementiert und getestet wurden, so dass später an der Funktionalität des WebServices nichts geändert werden muss, sondern nur an den Schnittstellen der Weboberfläche.

4.3 Präsentationsschicht

Nachdem die Funktionalität des Backends sichergestellt worden ist, konnte mit der Entwicklung der Präsentationsschicht begonnen werden. Die Blazor Server App bestehen aus mehreren Komponenten, welche modular aufgebaut sind. Es handelt sich hierbei um eine Single- Page- Applikation.

Das Frontend basierte zunächst auf einer Blazor-Server-App, wie man sie mittels des Visual Studio erstellen kann. Später wurde die BatteryCellApp zu einer Bibliothek geändert, welche in den „App- Content- Frame“ eingebunden wird. Der App- Content- Frame wird in Kapitel 4.3.6 besprochen.

4.3.1 Einbinden des Connectors

Der Connector und das Domainmodel werden als NuGet- Paket in die App eingebunden. Hierdurch erhält sie deren Funktionalität. Um die Methoden des Connectors auch ansprechen zu können, wurde zunächst ein Service erstellt, welcher dessen Methoden anspricht. Dieser Service wird dann im späteren Verlauf für das Abfragen und Abspeichern der Daten verwendet.

4.3.2 Erstellen der Pages

Um die Daten zu visualisieren, werden in Blazor sogenannte Pages verwendet. Diese sind in einen HTML- und einen Codebereich unterteilt. Im Codebereich werden hierfür die Variablen und Funktionen definiert, welche dann im HTML genutzt werden können. Ebenfalls können hierbei die Service- Funktionen genutzt werden, welche man über Dependency- Injection einbindet, siehe Abbildung 14.

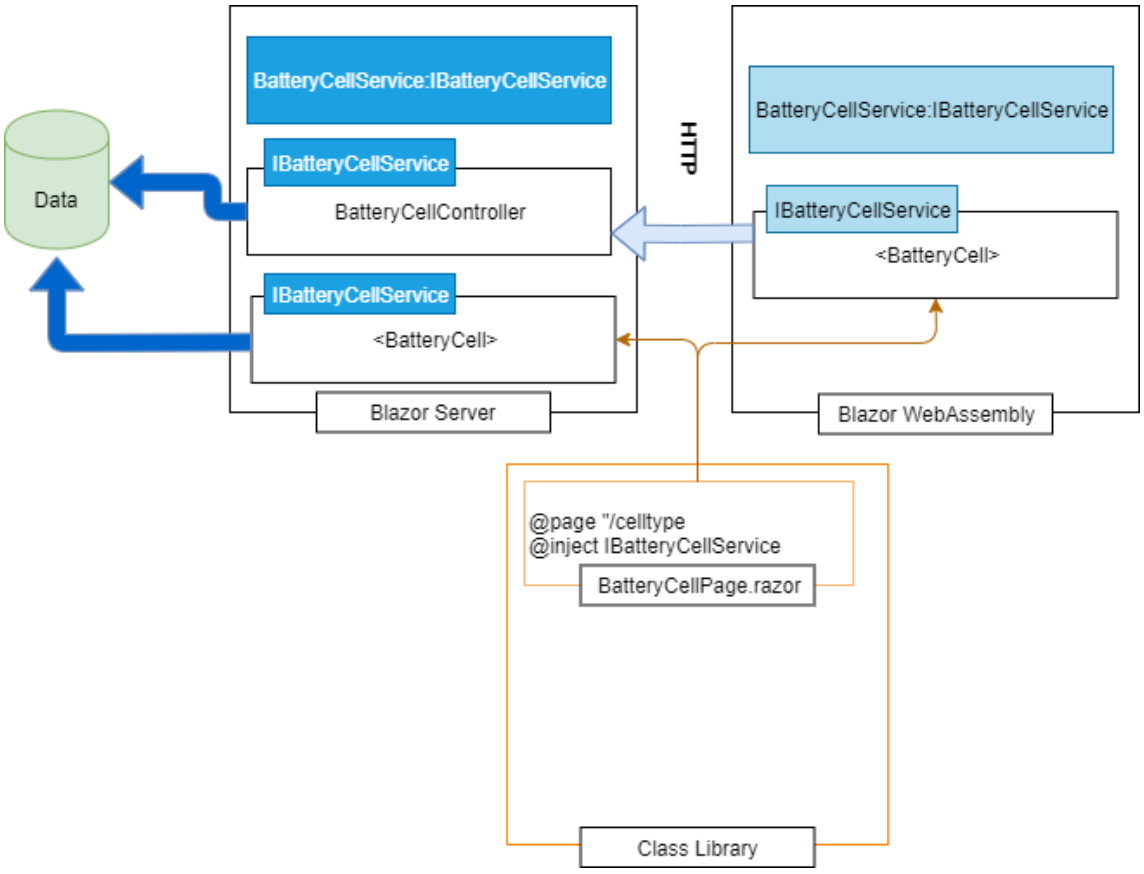


Abbildung 14: Einbinden von Services in Blazor (vgl. [22])

4.3.3 Entscheidung für Telerik

Zunächst wurde für die Darstellung des Datagrids und der Modal- Komponenten eine Open- Source- Lösung namens Blazorise verwendet. Diese hätte die für dieses Projekt ausreichende Funktionalität zur Verfügung gestellt. Da allerdings dies Projekt nur ein kleiner Teil ist und in anderen Projektteilen z.B. die Visualisierung der Daten in Charts benötigt werden, wurde sich für die kostenpflichtige, aber umfangreiche Technologie Telerik bevorzugt. Hierfür mussten die bereits entwickelten Komponenten abgeändert bzw. entfernt werden.

4.3.4 Visualisieren der Daten

Zunächst wurde die Darstellung der Werte aus der Datenbank innerhalb eines Datagrids (Datengitter) erstellt, wie diese genau verwendet wird, ist in Kapitel 5.2.2 beschrieben.

Die Komponente in der Anwendung, sieht dann aus wie Abbildung 15 dargestellt.

Abbildung 15: Grid- Komponente

Zusätzlich wird noch die Modal- Komponente verwendet, welche bei Telerik Window-Component genannt wird. Hierbei handelt es sich um Fenster, welche innerhalb der Seite geöffnet werden können, um spezielle Nutzereingaben zu fordern. Innerhalb dieses Projekts werden sie genutzt, um das Löschen, Editieren und Erstellen von Einträgen darzustellen. Das Ganze sieht aus wie in den folgenden Abbildungen dargestellt.

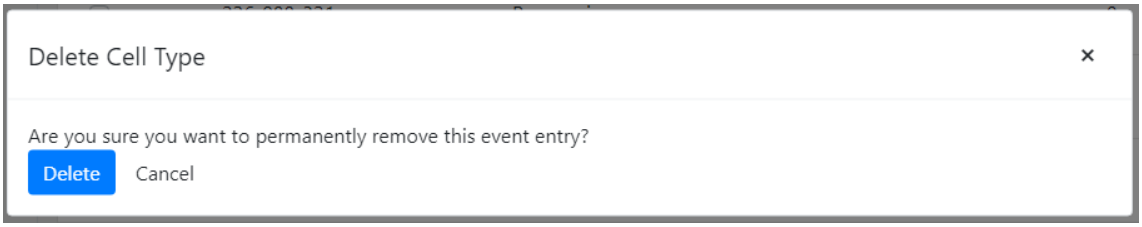


Abbildung 16: Delete- Kontext im Modal

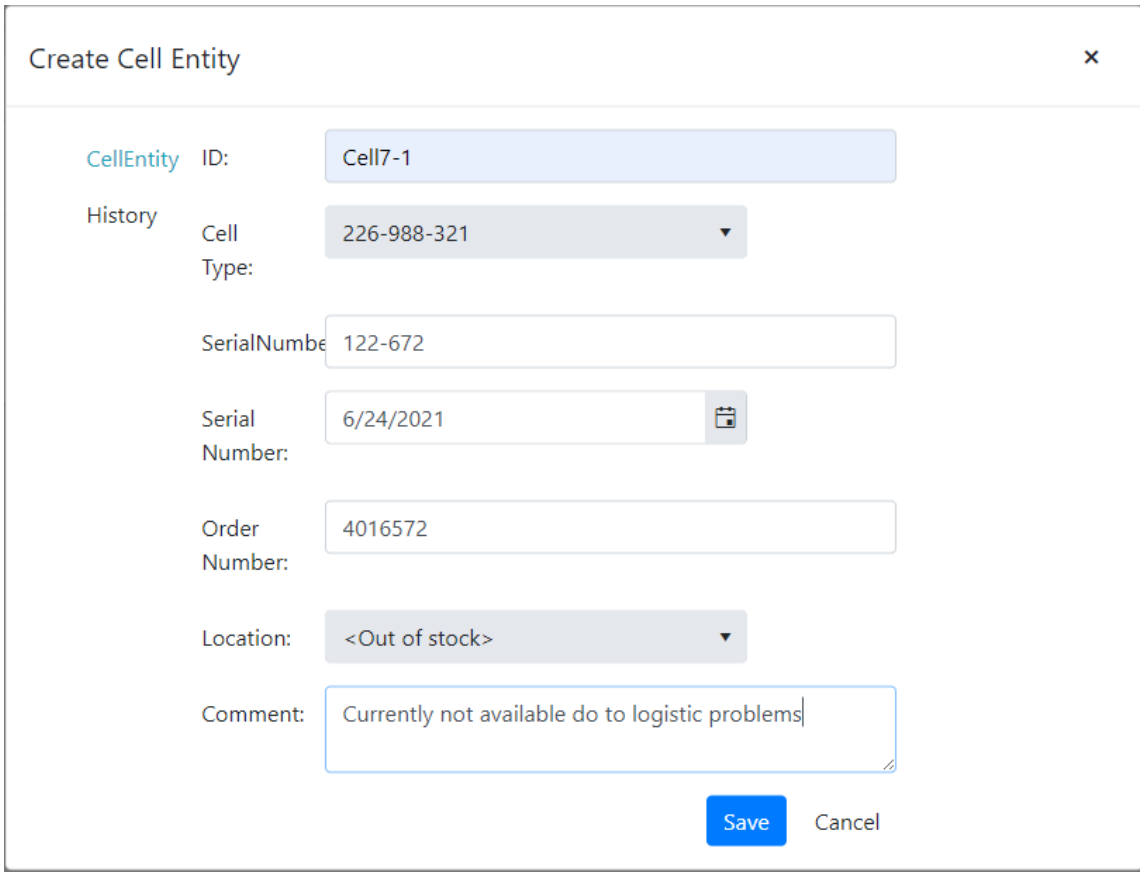


Abbildung 17: Create- Kontext im Modal

4.3.5 Methoden

Zusätzlich zur Visualisierung der Daten mussten auch die entsprechenden Methoden, für das Erstellen, Editieren, Entfernen entwickelt werden. Auch werden Methoden für das Öffnen und Schließen des Modals benötigt, sowie die Funktionalität für die SearchBar. Je nachdem, ob die Daten durch diese Aktionen verändert wurden, muss auch das Datagrid aktualisiert werden.

4.3.6 App- Content- Frame

Da es sich um ein großes Projekt handelt, wurde ein Rahmen entwickelt, welcher AppContentFrame heißt. Dieser fungiert als eine Art Gerüst, in welches die Bibliotheken, wie beispielsweise die BatteryCellApp eingebunden werden können und enthält eine Schnittstelle für das Löschen mehrerer Einträge, das Erstellen neuer Einträge sowie für das Filtern der Elemente.

Der AppContentFrame wurde entwickelt, damit die Weboberfläche einen einheitlichen Look erhält und darin nur die einzelnen Komponenten geladen werden müssen. Dies verhindert, dass die unterschiedlichen Anwendungen jeweils anders aussehen und damit gegebenenfalls den Nutzer verwirren. Außerdem muss dann bei späteren Designänderungen nur dieser AppContentFrame angepasst werden.

4.4 Deployment- Prozess

Nach dem die Anwendung voll funktionsfähig ist, konnte mit dem Deployment in der Azure Cloud begonnen werden.

Für das Deployment der Anwendung wurde AzureDevOps verwendet. Hierdurch können die entwickelten Komponenten in das DevOps eigene Git eingepflegt werden. Auch sind hier über die Azure Pipelines sowohl Continous Integration- als auch Continous Delivery Methoden verankert, welche die Projekte auf ihre Funktionalität überprüfen, beispielsweise über den ReSharper und SonarQube. Sollten die Tests erfolgreich sein, wird die Anwendung dann dementsprechend beispielsweise auf einem Webservice deployed oder in ein NuGet- Paket gebunden und zur Verfügung gestellt.

5 Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung der Anforderungen auf Basis der bereits eingeführten Technologien. Es werden exemplarisch Teile der Implementation erläutert, um die jeweiligen Vorzüge der verwendeten Technologien zu verdeutlichen. Im Folgenden werden nur die wichtigsten Ausschnitte der Komponenten besprochen und dargestellt.

5.1 Anwendungsschicht

Die Anwendungsschicht unterteilt sich in sechs Projekte, welche nun im Detail erläutert werden.

5.1.1 Datamodel

Im Datamodel befinden sich die Entitätsklassen, welche für die Generierung der Datenbank benötigt werden. Außerdem werden diese Entitätsklassen verwendet, um die Struktur der Daten für den Mapper deutlich zu machen.

Es werden partielle Klassen verwendet. Diese ermöglichen Erweiterbarkeit und verhindern, dass die Klassen durch das nächste Reverse- Engineering rückgängig gemacht werden. Da in diesem Projekt viele Entitätsklassen verwendet werden, welche teilweise sehr viele Attribute enthalten, wird im Folgenden nur beispielhaft eine Implementierung für eine Entitätsklasse, dargestellt.

```
[Table("CellEntity")]
    public partial class CellEntity
    {
        [Key]
        public Guid Id { get; set; }
        [Required]
        public string CustomId { get; set; }
        public Guid CellTypeId { get; set; }
        public string SerialNumber { get; set; }
        public DateTime? DeliveryDate { get; set; }
        public string OrderNumber { get; set; }
        public string Location { get; set; }
        public DateTime TimeStamp { get; set; }
        public string Comment { get; set; }
        [ForeignKey(nameof(CellTypeId))]
        [InverseProperty("CellEntities")]
        public virtual CellType CellType { get; set; }
    }
```

Codeteil 2: Entitätsklasse der CellEntity

Neben den Entitätsklassen muss auch der Datenbankkontext erstellt werden, welcher für die Zugriffe auf die Daten, welche sich innerhalb der Datenbank befinden, ermöglicht.

Aufgrund der Komplexität des Datenbankkontextes wird hier nur ein kurzes Beispiel zur Struktur dessen vorgestellt.

```
public partial class BatteryCellDbContext : DbContext
    {
        public BatteryCellDbContext(DbContextOptions<BatteryCellDbContext>
options)
            : base(options)
        {
        }

        public virtual DbSet<CellEntity> CellEntities { get; set; }
        public virtual DbSet<CellType> CellTypes { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.HasAnnotation("Relational:Collation", "Latin1_Gen-
eral_CI_AS");

            modelBuilder.Entity<CellEntity>(entity =>
            {
                entity.Property(e => e.Id).ValueGeneratedNever();

                entity.HasOne(d => d.CellType)
                    .WithMany(p => p.CellEntities)
                    .HasForeignKey(d => d.CellTypeId)
                    .OnDelete(DeleteBehavior.ClientSetNull)
                    .HasConstraintName("FK_CellEntity_CellType");
            });

            OnModelCreatingPartial(modelBuilder);
        }

        partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
    }
```

Codeteil 2: Beispiel des Datenbankkontextes

Hierbei wird eine DbSet- Eigenschaft für jede Entitätsmenge (CellEntities, CellTypes) erstellt sowie der Verweis der Zellentität auf den Zelltyp mit seiner Kardinalität angelegt (vgl. [17]).

5.1.2 Erstellen der Datenstruktur in der Datenbank

Damit Entity Framework Zugriff auf die Datenbank erhält, muss der Kontext noch in der Startup- Datei definiert werden.

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<BatteryCellDbContext>(options => options.UseSqlSer-
ver(Configuration.GetConnectionString("BatteryCellDatabase")));

        }
```

Codeteil 3: Hinzufügen des Datenbankkontextes

Zusätzlich muss die BatteryCellDatabase noch in der appsettings.json angelegt werden. Sobald dies geschehen ist, überprüft Entity Framework, ob die entsprechende Datenbank existiert, ansonsten wird versucht, diese zu erstellen (vgl. [17]).

5.1.3 Domainmodel

Im Folgenden wird beispielhaft die Implementierung des Domainmodels des Zelltyps gezeigt, hierbei wird das Interface ISerializable implementiert, damit die Objekte an die REST API übergeben werden.

```
[Serializable]
    public class CellType : ISerializable
    {

        public CellType()
        {
            Supplier supplier = new Supplier();
              …
        }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Usage", "CA1801:Review unu-
sed parameters", Justification = "Required parameter for .NET serialization.")]
        private CellType([NotNull] SerializationInfo info, StreamingContext context)
        {
            if (info == null)
                throw new ArgumentNullException(nameof(info));

            Id = info.GetGuid(nameof(Id));
              …
            Supplier = info.GetValue<Supplier>(nameof(Supplier));
        }

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

            info.AddValue(nameof(Id), Id);
            info.AddValue(nameof(Supplier), Supplier);
        }
        public Guid Id { get; set; }
         …
        [Required]
        [MinLength(5, ErrorMessage = "Length has to be 5 or more")]
        [MaxLength(100, ErrorMessage = "Length has to be 100 or less")]
        public string SupplierAddress
        {
            get => Supplier.Address;
            set => Supplier.Address = value;
        }
       …

    }
```

Codeteil 4: Aufbau des Domainmodells des Zelltyps

Im Domänenmodell befinden sich ebenfalls die Informationen, welche für die Validierung der Daten im Frontend wichtig sind. Beispielsweise wie eine E- Mail korrekt definiert ist, wie in folgendem Beispiel zu sehen.

```
[Required(ErrorMessage = "Enter a valid E-mail")]
        [RegularExpression("[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}")]
        public string SupplierEmail
        {
            get => Supplier.Email;
            set => Supplier.Email = value;

        }
```

Codeteil 5: Validierung der E- Mail

5.1.4 Webservice

Der Webservice ist für die gesamte Geschäftslogik und das Bereitstellen der REST API zuständig, aber auch für das Mapping der Modelle. Im folgenden Codeteil sieht man eine beispielhafte Konfiguration des Automappers zum Umwandeln der Modelle.

```
public class MappingProfileBatteryCell : MappingProfile
    {
        public MappingProfileBatteryCell([NotNull] ILocalizer localizer) : base(loca-
lizer)
        {
            MapCellType();
              …
        }
        private void MapCellType()
        {
            CreateMap<CellTypeDto, CellType>();
            CreateMap<CellType, CellTypeDto>();
        }
              …
    }
```

Codeteil 6: Mapping der Modelle via Automapper

Damit das Mapping besonders anschaulich dargestellt ist, wurde das Mapping zwischen dem Daten- und Domainmodell des Zelltyps gewählt. Hierfür benötigt man zunächst eine Klasse, welche von der Basisklasse `Profile` erbt. Die Konfiguration des Mapping wird dann im Konstruktor dieser Klasse vorgenommen. Hierbei werden die Properties der Objekte, welche bereits den gleichen Namen und Datentyp haben, automatisch korrekt gemapped. Sollte es Abweichungen zwischen den Modellen geben, können diese über die Methode `ForMember` dementsprechend angepasst werden.

Die Definition der REST API erfolgt über eine Kontroller Klasse, die Definition sieht aus wie folgt:

```
[ApiController]
    [Route(ApiBatteryCellPath)]
    public partial class BatteryCellController : ControllerBase
    {
        private const string ApiBatteryCellPath = "/api/batterycell";

        private readonly CellTypeService _cellTypeService;
              …
        public BatteryCellController(ILogger<BatteryCellController> logger, CellType-
Service cellTypeService, …)
        {
            Logger = logger;
            _cellTypeService = cellTypeService;
            …
        }

        public ILogger<BatteryCellController> Logger { get; }
    }
```

Codeteil 7: Definition des REST API Kontrollers

Die im Kontroller initialisierten Services regeln den Datenzugriff auf die Datenbank über den Datenbankkontext und Entity Framework. Hierbei werden zum Zelltyp über das Keyword `Include` noch die von ihm abhängenden Kind- Elemente angehängt, wie im nachfolgenden Beispiel gezeigt:

```
public class CellTypeService : BatteryCellEntityService<CellTypeDto>
    {
        private readonly IMapper _mapper;
        private readonly ILogger<CellType> _logger;

        public CellTypeService(BatteryCellDbContext dbContext, IMapper mapper, ILog-
ger<CellType> logger) : base(dbContext)
        {
            _mapper = mapper;
            _logger = logger;
        }

              …
        public async Task<BatteryCellActionResult<CellTypeDto>> GetCellTypeWithChild-
renAsync(Guid id)
        {
            var entity = await DbContext.CellTypes.Include(x => x.ChargeCharacteris-
tic).Include(x => x.DischargeCharacteristic).Include(x => x.Supplier).Include(x =>
x.StorageCharacteristic).Include(x=>x.PerformanceSpecification).Include(x=>x.Physi-
calSpecification)

                .SingleOrDefaultAsync(item => item.Id == id).ConfigureAwait(false);
              …

            return Ok(_mapper.Map<CellTypeDto>(entity));
        }
    }
```

Codeteil 8: Services zum Datenzugriff auf Datenbank

Die Basisklasse Kontroller enthält Kindklassen, welche jeweils einer Entität zugeordnet sind und die jeweilige REST Schnittstelle definieren, welche dann den Service

ansprechen können, um Daten abzufragen oder zu erstellen. Hierbei wird auch der Return- Type festgelegt, welcher im Erfolgs- oder Fehlerfall zurückgegeben wird. Im Folgenden ist die GET- Schnittstelle dargestellt, welche den Zelltyp mit seinen Kind- Elementen zurückgibt.

```
[HttpGet("celltypewithchildren")]
        [Produces("application/json")]
        [ProducesResponseType(typeof(IList<CellType>), StatusCodes.Status200OK)]
        [ProducesResponseType(typeof(IList), StatusCodes.Status204NoContent)]
        public async Task<ActionResult<IList<CellType>>> GetCellTypesWithChildrenA-
sync()
        {
            var result = await _cellTypeService.GetCellTypesWithChildrenAsync().Confi-
gureAwait(false);

            return result;
        }
```

Codeteil 9: Definition der REST API Schnittstellen

5.1.5 Dokumentation der Schnittstellen

Für die Dokumentation der Schnittstellen mittels Swashbuckle werden im Code XML-Kommentare und Annotations- Attribute verwendet, wie im folgenden Codeteil zu sehen.

```
/// <summary>
        /// Returns all CellTypes.
        /// </summary>
        /// <returns>List of CellTypes</returns>
        /// <response code="200">Returns a List of CellTypes</response>
        /// <response code="204">Returns empty List if no CellTypes are
found.</response>
        [HttpGet("celltypes")]
        [Produces("application/json")]
        [ProducesResponseType(typeof(IList<CellType>), StatusCodes.Sta-
tus200OK)]
        [ProducesResponseType(typeof(IList), StatusCodes.Status204NoCon-
tent)]
        public async Task<ActionResult<IList<CellType>>> GetCellType-
sAsync()
        {
            var result = await _cellTypeService.GetCellTypesAsync().Config-
ureAwait(false);

            return result;
        }
```

Codeteil 10: Swashbuckle Implementierung

Die daraus entstehende Zusammenfassung wird in Kapitel 4.2.4.3 exemplarisch dargestellt.

5.1.6 Integration Tests für den Webservice

Die Integrationstests für den Webservice sehen aus wie in folgendem Codebeispiel dargestellt.

```
public class BatteryCellControllerTests
    {
        private static readonly IList<CellEntityDto> EntityList = new
List<CellEntityDto>();

        [Fact]
        public async Task GetCellEntitiesAsyncTest()
        {
            var cellEntityService = new Mock<ICellEntityService>(MockBehav-
ior.Strict);
            cellEntityService.Setup(mock => mock.GetCellEntitiesAsync())
                             .ReturnsAsync(new BatteryCellActionRe-
sult<IList<CellEntityDto>>(EntityList, new Bat-
teryCellHttpStatus(HttpStatusCode.OK, ""), "",

new ResponseParameterDictionary()));
            var testee = new BatteryCellController(null, null, null, cel-
lEntityService.Object, null, null, null, null, null);

            var response = await testee.GetCellEntitiesAsync();

            cellEntityService.VerifyAll();
            Assert.NotNull(response.Result);
            Assert.True(response.Result is ObjectResult);
            Assert.Same(EntityList, ((ObjectResult) response.Re-
sult).Value);
        }
    }
```

Codeteil 11: Integrationstests am Webservice

5.1.7 Connector

Die Services, welcher der Connector zur Verfügung stellt, sehen aus wie die folgende Methode zum Zugriff auf den Zelltyp mit seinen Kind- Elementen, welche schon aus den vorangegangenen Beispielen bekannt ist:

```
internal partial class BatteryCellService
    {
        private const string ApiCellTypePath = "/api/bat-
terycell/celltypeswithchildren";

        public async Task<IList<CellType>> GetCellTypesWithChildrenAsync()
        {
            var request = new RestRequest(new Uri($"{ApiCellTypePath}"
, UriKind.Relative));
            var response = await _connector.ExecuteGetA-
sync<IList<CellType>>(request).ConfigureAwait(false);

            return response.Data;
        }
    }
```

Codeteil 12: Definition der Connector Services

Zur Verfügung gestellt werden hier die CRUD- Operationen für jede der Entitäten der Datenbank.

5.1.8 Unit Tests für den Connector

Zunächst einmal muss eine Klasse definiert werden, welche aussieht wie folgt:

```
public partial class BatteryCellTest : TestBase
    {
        private readonly ITestOutputHelper _output;
        protected IBatteryCellService BatteryCellService { get; }

        public BatteryCellTest(ITestOutputHelper output)
        {
            _output = output;
            BatteryCellService = ServiceProvider.GetService<IBattery-
CellService>();
        }
    }
```

Codeteil 13: BatteryCellTest Klasse

Diese erbt ihre Eigenschaften von der Basis Klasse `TestBase,` welche definiert ist wie folgt:

```
public class TestBase : IDisposable
    {
        public TestBase()
        {
            var collection = new ServiceCollection();
            collection.AddBatteryCellWebServiceConnector(CreateConfigura-
tion());
            ServiceProvider = collection.BuildServiceProvider();
        }
        protected IServiceProvider ServiceProvider { get; }
        private static IConfiguration CreateConfiguration()
        {
            return new ConfigurationBuilder()
                .AddJsonFile("appsettings.Local.json", true)
                .Build();
        }
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
            }
        }
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
```

Codeteil 14: BatteryCellTest Klasse

Sie nutzt die Konfiguration, welche in der `appsettings.Local.json` enthalten ist um einen Provider für den Test zu erstellen.

Die BatteryCellTest Klasse enthält Kind- Elemente, welche jeweils die einzelne Funktionalität eines Endpoints testen. Für die Rückgabe jeder Funktion gibt es einen eigenen Test, der dessen Funktionalität überprüft. Im Folgenden ist der Test für den Endpoint `/api/batterycell/celltypeswithchildren` zu sehen.

```
[Theory]
        [InlineData("94cbcf66-ac55-4489-9931-dcea213dda32")]
        public async Task GetCellTypesWithChildrenTest(string Id)
        {
            await Before();

            var cellTypes = await BatteryCellService.GetCellTypesWithChil-
drenAsync().ConfigureAwait(false);
            Assert.NotNull(cellTypes);
            Assert.Equal("TestNameCellType", cellTypes.FirstOrDefault(x =>
x.Id == Guid.Parse(Id)).Name);
            Assert.Equal("TestSaveSupplierEdit", cellTypes.FirstOrDefault(x
=> x.Id == Guid.Parse(Id)).Supplier.Name);
        }
```

Codeteil 15: UnitTest GetCellTypesWithChildren

Hierbei wird vor dem Ausführen der Tests durch die before- Methode für einen konsistenten Zustand der Datenbank als Vorbereitung für den Test gesorgt, welcher wie folgt definiert ist:

```
private async Task<BatteryCellDbContext> before()
        {
            var options = new DbContextOptionsBuilder<BatteryCellDbCon-
text>()
                .Options;
            var databaseContext = new BatteryCellDbContext(options);
            databaseContext.Database.EnsureCreated();
            await RemoveDatabaseContext(databaseContext);

            databaseContext.ChargeCharacteristics.Add(_charge);
            databaseContext.DischargeCharacteristics.Add(_discharge);
            databaseContext.PerformanceSpecifications.Add(_performance);
            databaseContext.PhysicalSpecifications.Add(_physical);
            databaseContext.StorageCharacteristics.Add(_storage);
            databaseContext.Suppliers.Add(_supplier);
            databaseContext.CellTypes.Add(_cellType);
            await databaseContext.SaveChangesAsync();

            return databaseContext;
        }
```

Codeteil 16: before- Methode

Der oben genannte Test fragt nach einem der Zelltypen, welcher in der Datenbank enthalten ist und überprüft, ob dieser existiert, sein Name korrekt ist und auch sein Supplier-Objekt korrekt angehängt wurde.

5.2 Präsentationsschicht

5.2.1 Service

Wie bereits erwähnt wird der Connector zusammen mit dem DomainModel als NuGet-Paket in das Frontend eingebunden. Zunächst musste erst einmal ein Service erstellt werden, welcher die Funktionen des Connectors für die App nutzbar macht:

```
public partial class BatteryCellDataAccess
    {
        private readonly IBatteryCellService _batteryCellService;

        public BatteryCellDataAccess(IBatteryCellService bat-
teryCellService)
        {
            _batteryCellService = batteryCellService;
        }
    }
```

Codeteil 17: BatteryCellDataAccess

Dieser DataAccess Service enthält die CRUD- Methoden für den Zugriff auf die Schnittstellen des Connectors.

```
public partial class BatteryCellDataAccess
    {
        public async Task<IList<CellType>> GetCellTypesAsync()
        {
           return await _batteryCellService.GetCellTypesWithChildrenAsync().Configu-
reAwait(false);
        }

        public async Task<CellType> SaveCellTypeWithChildrenAsync(CellType cellType)
        {
            return await _batteryCellService.SaveCellTypeWithChildrenAsync(cell-
Type).ConfigureAwait(true);
        }

        public async Task DeleteCellTypeAsync(Guid id)
        {
            await _batteryCellService.DeleteCellTypeAsync(id).ConfigureAwait(true);
        }

        public async Task<CellType> GetCellTypeAsync(Guid id)
        {
            return await _batteryCellService.GetCellTypeAsync(id).ConfigureA-
wait(true);
        }

    }
```

Codeteil 18: BatteryCellDataAccess.CellType

Über die Dependency- Injection von Blazor kann dieser Service beliebig in jede Komponente eingebunden und seine Methoden genutzt werden, indem man ihn mittels des @using Statements einbindet und mittels @inject initalisiert.

```
@using Systecs.TF.BatteryCell.Services

@inject BatteryCellDataAccess CellTypeService
```

Codeteil 19: using and inject

5.2.2 Pages

Nachdem die Daten nun mittels der Services abgefragt werden können, ist es möglich, mit der Visualisierung der Zelltypen und Zellentitäten zu beginnen.

Hierfür wird die Grid- Komponente von Telerik verwendet. Dieser übergibt man eine Liste der Daten, welche man anzeigen möchte.

```
<TelerikGrid Data="_cellTypes" EditMode="@GridEditMode.Inline" Sortable="true"
SelectionMode="@GridSelectionMode.Multiple" SelectedItems="@SelectedCellTypes"
SelectedItemsChanged="@((IEnumerable<CellType> cellTypeList) => OnSelect(cell-
TypeList))"
             Pageable="@(_cellTypes?.Count > pageSize)" PageSize="@pageSize"
             OnRowRender="@OnRowRenderHandler"
             Resizable="true" Reorderable="true"
             Class="no-scroll">
    <GridColumns>
        <GridCheckboxColumn/>
        <GridColumn Field="@(nameof(CellType.Name))" Title="Name" Group-
able="false"/>
        <GridColumn Field="@(nameof(CellType.SupplierName))" Title="Supplier"
Groupable="false"/>
        <GridColumn Field="@(nameof(CellType.PhysicalSpecificationConstruc-
tion))" Title="Form" Groupable="false"/>
        <GridColumn Field="@(nameof(CellType.PerformanceSpecificationCapacityNo-
minal))" Title="Capacity (nominal)" Groupable="false"/>
        <GridCommandColumn Width="70px">
            <GridCommandButton Command="EditCellType" OnClick="@OnEditCellType"
Class="custom-command-button mr-2">
                <i class="fas fa-edit"></i>
            </GridCommandButton>
            <GridCommandButton Command="EditCellType" OnClick="@OnDeleteCell-
Type" Class="custom-command-button mr-2">
                <i class="fas fa-trash"></i>
            </GridCommandButton>
        </GridCommandColumn>
    </GridColumns>

</TelerikGrid>
```

Codeteil 20: Telerik- Grid CellType

In diesem Grid wird der Name des Zelltyps, der Name des Suppliers, die Konstruktionsart als auch die Nominal Kapazität ausgegeben. Zusätzlich gibt es noch Buttons, welche zum Editieren und Löschen des jeweiligen Zelltyps verwendet werden können.

5.2.3 Modal- Komponente

Die Buttons für das Erstellen, Editieren und Löschen öffnen jeweils eine Modal- Komponente. Diese Modals sind kleine Fenster innerhalb der Seite.

Hierfür werden der Komponente über Parameter die nötigen Objekte und Methoden übergeben.

```
private async Task OnCreateCellType()
    {
        await cellTypeModalRef.Show(ModalType.Create, new CellType());

    }
```

Codeteil 21: Erstellen eines neuen Zelltyps

Parameter werden in Blazor übergeben, indem man in der Komponente den Parameter-Tag verwendet.

```
[Parameter]
public EventCallback<(ModalType, CellType)> ModalCallback { get; set; }
```

Codeteil 22: Erstellen eines neuen Zelltyps

Innerhalb der Modal- Komponente wird erst einmal entschieden, ob es sich um ein Create, Update, Delete oder Delete Multiple handelt. Je nachdem wird der Title des Modals angepasst. Sollte es sich um ein Delete oder Delete Multiple handeln, wird der Nutzer gebeten, zu bestätigen, dass er die Einträge tatsächlich löschen möchte.

Sollte das Löschen bestätigt werden, wird die dazugehörige Delete- Methode innerhalb der Modal- Komponente aufgerufen und das Modal aktualisiert. Das InvokeAsync benachrichtigt hierbei die Komponente, dass es Änderungen gab und aktualisiert die Komponente.

```
private async Task Delete()
    {
        Hide();

        await _batteryCellDataAccess.DeleteCellTypeAsync(ParentId);
        await ModalCallback.InvokeAsync((_modalType, _cellType));

    }
```

Codeteil 23: Delete- Methode

Telerik ermöglicht es mit der Multiselect Option mehrere Listenelemente auszuwählen. Sollte der Nutzer mehrere Elemente ausgewählt haben, hat er die Möglichkeit, den Delete(X)- Button zu drücken, um die ausgewählten Einträge zu löschen. Dafür wird die Liste der ausgewählten Elemente an die DeleteMultiple- Methode übergeben.

```
public async Task DeleteMultiple()
    {
        Hide();

        foreach (var selectedCellType in SelectedCellEntities)
        {
            await _batteryCellDataAccess.DeleteCellTypeAsync(selectedCellType.Id);
        }

        await ModalCallback.InvokeAsync((_modalType, _cellType));

    }
```

Codeteil 24: DeleteMultiple- Methode

5.2.4 Custom- Sidebar- Komponente

Sollte es sich um ein Create oder Update handeln, wird die Seitennavigations- Komponente, sowie die Forms- Eingabe- Komponente geladen.

Die Navigationskomponente ist aufgebaut wie folgt:

```
<ol class="customSidebar">
                        @foreach (var item in Data)
                        {
                            <li @onclick="@(() => SelectedItem = item)"
                                class="k-drawer-item @GetSelectedItemClass(item)"
style="white-space: nowrap">
                                @item.Text
                            </li>
                        }

                    </ol>
```

Codeteil 25: Sidebar- Komponente

Sie bezieht die Namen ihrer Einträge aus einem Container und öffnet, je nach Auswahl die dazu passende Forms- Komponente und übergibt dieser die benötigten Parameter und Methoden.

```
@if (SelectedItem?.Text == "General")
                            {
                                <CreateGeneral _cellTypeEditContext="_cellTypeEdit-
Context" UpdateValidation="@UpdateValidation" cellTypeEdit="_cellType" Save="@Save"
Hide="@Hide" isValid="@isValid" _modalType="_modalType"></CreateGeneral>

                            }
```

Codeteil 26: Aufrufen der Forms- Komponente für General

5.2.5 Forms- Komponente

Die Forms- Komponenten enthalten die Eingabefenster für den Nutzer und sind ebenso für die Eingabevalidierung zuständig.

```
<TelerikForm EditContext="_cellTypeEditContext">
        <FormValidation>
            <DataAnnotationsValidator></DataAnnotationsValidator>
        </FormValidation>
        <FormItems>
            <FormItem Class="row form-group d-flex">
                <Template>
                    <FormItemTemplate Label="Name">
                        <CustomInput>
                            <TelerikTextBox Enabled="@IsCreateMode()" Value="@cell-
TypeEdit.Name" ValueExpression="@(() => cellTypeEdit.Name)" ValueChanged="@(async
(string v)  => { cellTypeEdit.Name =v; await UpdateValidation.InvokeAsync(); })"
Id="Name" PlaceHolder="Enter Name..."/>
                        </CustomInput>
                        <CustomValidation>
                            <TelerikValidationMessage For="@(() => cell-
TypeEdit.Name)"/>
                        </CustomValidation>
                    </FormItemTemplate>
                </Template>
            </FormItem>
        </FormItems>
        <FormButtons>
            <TelerikButton ButtonType="ButtonType.Button" OnClick="@Save" Enab-
led="@isValid" Primary="true">Save</TelerikButton>
            <TelerikButton ButtonType="ButtonType.Button" OnClick="@Hide">Cancel</Te-
lerikButton>
        </FormButtons>

    </TelerikForm>
```

Codeteil 27: Beispiel für die Eingabe des Namens

Über den Tag Enabled wird der Save- Button deaktiviert, bis alle benötigten Werte eingetragen wurden und valide sind.

5.2.6 FormItemTemplate- Komponente

Die Strukturierung der Eingabe wird von der FormItemTemplate- Komponente übernommen. Diese sorgt dafür, dass die Eingabe immer gleich aufgebaut ist und auch die Validierungsbenachrichtigung unter dem Eingabefenster angezeigt wird, welches invalide ist.

```
<label for="@Label" class="col-sm-2 col-form-label">@(Label):</label>
<div class="col-sm-10">
    @CustomInput
</div>

<div class="col-sm-2"></div>
<div class="col-sm-10">
    @CustomValidation
</div>

@code {

    [Parameter]
    public string Label { get; set; }

    [Parameter]
    public RenderFragment CustomInput { get; set; }

    [Parameter]
    public RenderFragment CustomValidation { get; set; }


}
```

Codeteil 28: FormItemTemplate

5.2.7 Validierung der eingegebenen Werte

Die Validierung der eingegebenen Werte läuft über die UpdateValidation Methode, welche als Parameter aus der Modal- Komponente übergeben wurde. Sobald sich der eingegebene Wert innerhalb des Fensters ändert, wird die Methode UpdateValidation aufgerufen. Sie vergleicht gegen das Domain- Model, ob der eingegebene Wert valide ist.

```
private void UpdateValidation()
    {
        isValid = _cellTypeEditContext.Validate();

    }
```

Codeteil 29: Beispiel für die Eingabe des Namens

Sollte der Wert nicht valide sein, wird die passende ValidationMessage ausgegeben, welche im Domain- Model definiert ist. Im Folgenden ist die Validierung für den Zelltypnamen zu sehen. Hierbei wird überprüft, ob es sich um einen String handelt und seine Länge zwischen 3 und 20 Zeichen liegt.

```
        [Required(ErrorMessage = "Enter a valid Name!")]
         [MinLength(3, ErrorMessage = "The Name length has to be 3 or more")]
         [MaxLength(20, ErrorMessage = "The Name length has to be 20 or less")]

         public string Name { get; set; }
```

Codeteil 30: Validierung des Namens

5.2.8 Speichern in der Datenbank

Ist die Validierung korrekt, hat der Nutzer die Möglichkeit, alle Eingaben zu verwerfen oder zu speichern. Beim Drücken des Cancel Buttons wird das erstelle Objekt verworfen und das Modal- Fenster geschlossen. Beim Drücken des Save- Buttons wird die übergebene Save- Methode der Modal- Komponente verwendet, um das erstellte Objekt an das Backend zu senden.

```
private async Task Save()
    {
        isValid = _cellTypeEditContext.Validate();

        if (!isValid)
            return;

        Hide();
              …

        var saveEventEntry = await _batteryCellDataAccess.SaveCellTypeWithChildrenA-
sync(_cellType);

        await ModalCallback.InvokeAsync((_modalType, saveEventEntry));

    }
```

Codeteil 31: Speichern des Objekts in der Datenbank

5.2.9 Aktualisierung der Werte im Grid

Nachdem das Objekt an die Datenbank gesendet wurde, wird das neue Objekt in das Grid eingefügt. Dies erfolgt über eine eigene Komponente, um die Datenbankzugriffe besonders gering zu halten.

```
public static void Update(this CellType dest, CellType source)
        {
            if (dest != null && source != null)
            {
                dest.Name = source.Name;
                dest.SupplierName = source.SupplierName;
                dest.PhysicalSpecification.Construction = source.PhysicalSpecifica-
tion.Construction;
                dest.PerformanceSpecification.CapacityNominal = source.PerformanceSpe-
cification.CapacityNominal;
            }

        }
```

Codeteil 32: Aktualisieren des Data-Grid

5.2.10 Searchbar- Komponente

In der Searchbar- Komponente wird es ermöglicht, Filter für die Daten innerhalb des Grids zu vergeben. Hierfür gibt es eine eigene Komponente, welche aussieht wie folgt:

```
   private async Task OnSearch(string searchString)
    {
        searchString = searchString.Trim();

        var currentState = Grid.GetState();

        currentState.FilterDescriptors.Clear();

        var filterCollection = new FilterDescriptorCollection
        {
            new FilterDescriptor(COLUMN_CELL_TYPE_NAME, FilterOperator.Contains,
searchString)

        };
```

In dieser Komponente wird überprüft, ob der Name der Zelltypen mit dem übergebenen Suchbegriff übereinstimmt.

6 Zusammenfassung und Ausblick

Die innerhalb des Projekts gestellten Anforderungen wurden funktional umgesetzt, nur das Design muss noch an die Anforderungen der heutigen Zeit angepasst werden, um auch auf Mobilgeräten ansprechend auszusehen. Die erreichte Performanz der Anwendung ist sehr gut.

Das Projekt wurde ohne das Schreiben von JavaScript umgesetzt, was wiederum die Funktionalität und Leistungsfähigkeit des ASP.NET Frameworks verdeutlicht. Insgesamt stellt das ASP.NET Framework dem Entwickler viele Möglichkeiten zur Verfügung, welche das schnelle Entwickeln einer Webanwendung begünstigen. Allerdings ist dies mit einem höheren Einarbeitungsaufwand verbunden, da das Framework durchaus seine Eigenheiten aufweist. Durch das Einbinden von NuGet- Paketen können jederzeit neue Features zur Anwendung hinzugefügt werden.

Durch den Einsatz von Blazor Server bleiben die Transaktionsdaten geheim und es wird eine gute Performanz auch auf Thin- Clients erzielt. Sollte das Projekt auf den Einsatz mit vielen Nutzern umgestellt werden müssen, kann man auf Blazor WebAssembly zurückgreifen, ist dann aber den Unsicherheiten dieser Implementierung unterworfen.

Durch die Anbindung an Azure kann individuell auf neue Gegebenheiten z.B. bei der Leistung des Webservers reagiert werden, sollte die Seite beispielsweise doch von mehr Nutzern verwendet werden, als nur innerhalb des Unternehmens.

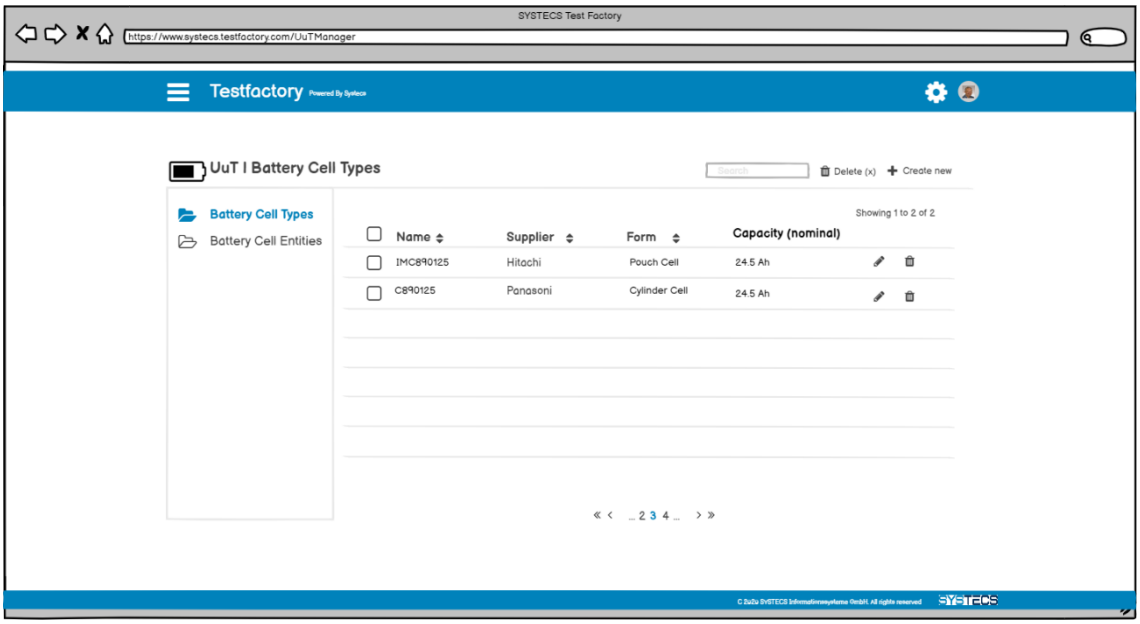
Durch die zahlreichen Integrations- und Unittests kann die Sicherheit der einzelnen Komponenten und Schnittstellen gewährleistet werden. Die Technologien ReSharper sowie Sonar Qube welche in DevOps eingesetzt werden, überprüfen und gewährleisten eine gute Codequalität des Projekts. Auch ist es möglich über das in DevOps enthaltene Git-Repository mit mehreren Entwicklern an diesem Projekt zu arbeiten.

Wie bereits angesprochen, müsste um die verschiedenen Bauformen der Batteriezelle korrekt in der Datenbank ablegen zu können, das Datenmodell noch dementsprechend überarbeitet werden. Durch Telerik ist es möglich, auch aufwendige Charts, beispielsweise über den Verlauf der Tests generieren zu lassen. Hierdurch sollte der Weiterentwicklung des Projekts nichts im Wege stehen.

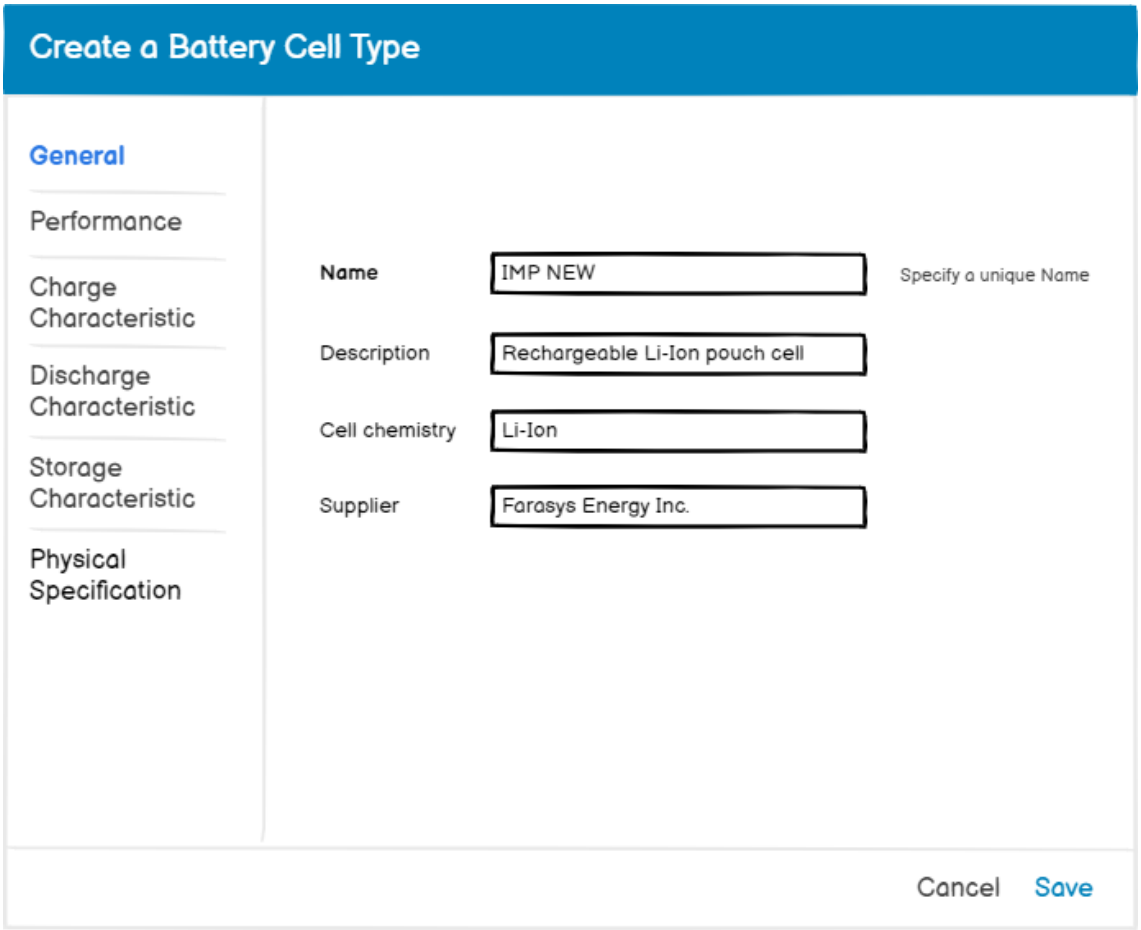
Anhang:

Anhang A.1 Datenmodell

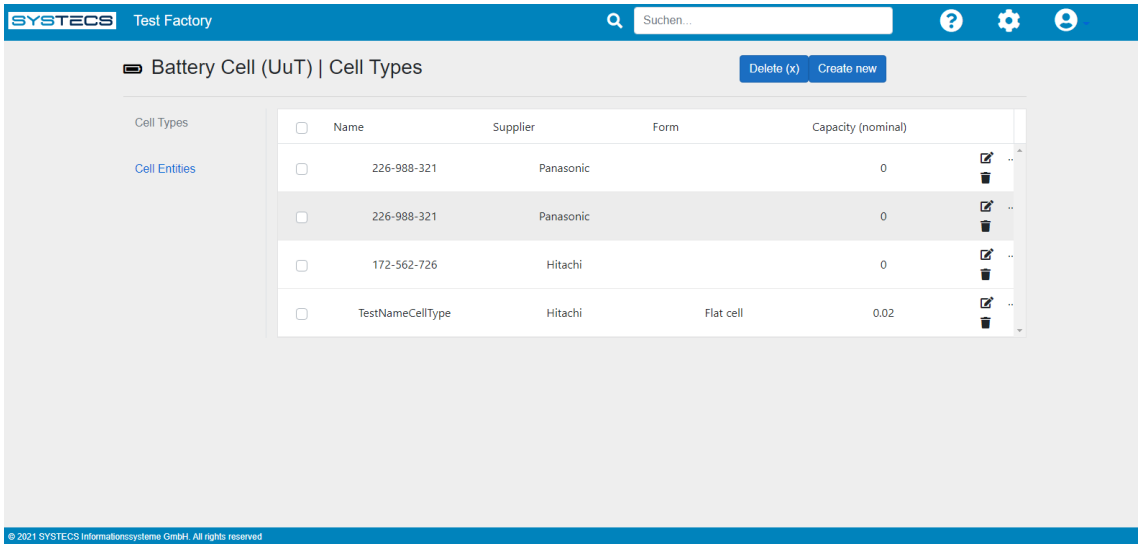
Anhang B.1 Prototyp Modalansicht Zelltyp



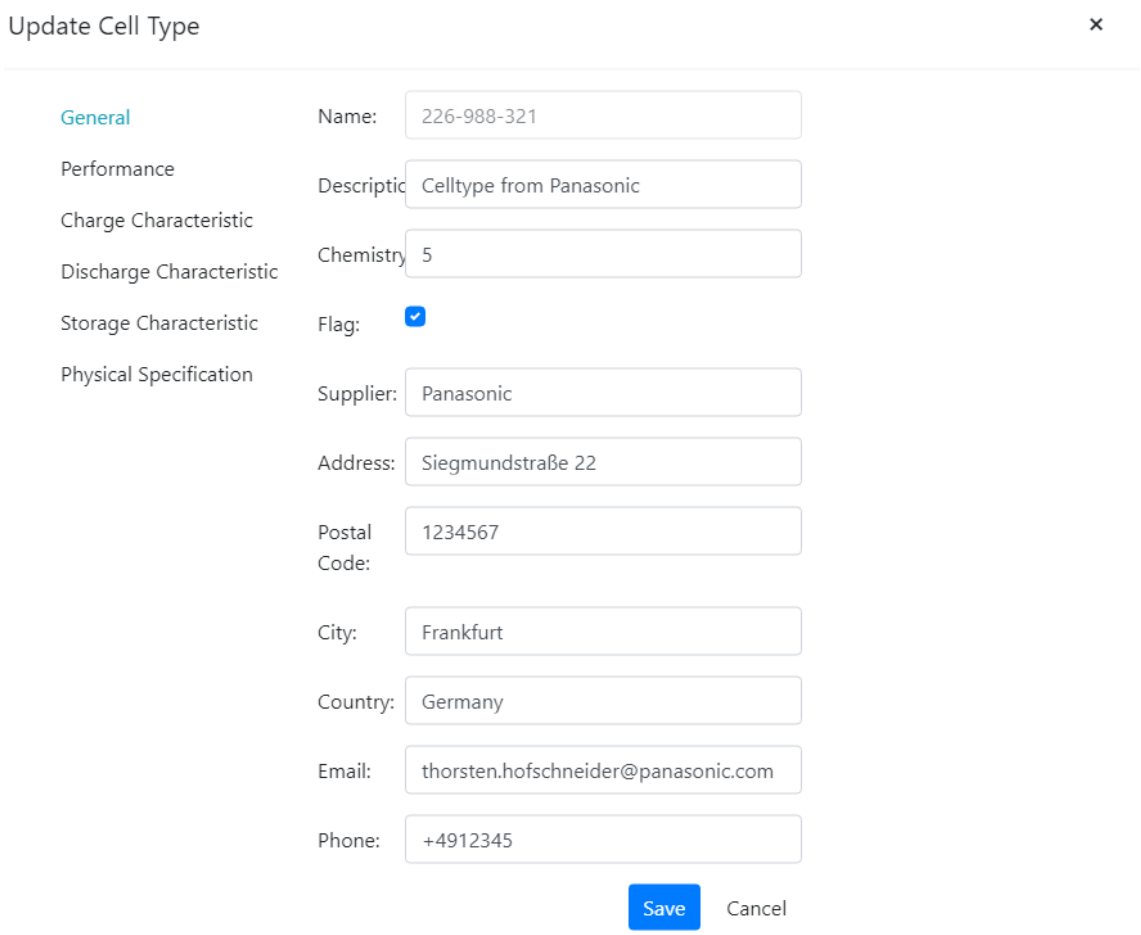
Anhang B.2 Prototyp Modalansicht Zelltyp



Anhang C.1 Entwickelte Anwendung Übersichtsseite



Anhang C.2 Entwickelte Anwendung Modalansicht Zelltyp



Literaturverzeichnis

[1] Microsoft Corporation, „Erste Schritte mit .NET Framework,“ Microsoft Corporation, 21 10 2020. [Online]. Available: https://docs.microsoft.com/de-de/dotnet/framework/get-started/. [Zugriff am 11 05 2021].

[2] Microsoft Corporation, „ASP.NET Security Architecture,“ Microsoft Corporation, 22 10 2014. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/aspnet/yedba920(v=vs.100)?redirectedfrom=MSDN. [Zugriff am 11 05 2021].

[3] Microsoft Corporation, „Einführung in .NET,“ Microsoft Corporation, 16 11 2020. [Online]. Available: https://docs.microsoft.com/de-de/dotnet/core/introduction. [Zugriff am 11 05 2021].

[4] R. A. S. L. Daniel Roth, „Einführung in ASP.NET Core,“ Microsoft Corporation, 17 4 2020. [Online]. Available: https://docs.microsoft.com/de-de/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0. [Zugriff am 11 5 2021].

[5] Microsoft Corporation, „Einführung in ASP.NET Blazor,“ Microsoft Corporation, 25 9 2020. [Online]. Available: https://docs.microsoft.com/de-de/aspnet/core/blazor/?view=aspnetcore-5.0. [Zugriff am 3 6 2021].

[6] R. N. Rick Anderson, „Einführung in Razor Pages in ASP.NET Core,“ Microsoft Corporation, 12 2 2020. [Online]. Available: https://docs.microsoft.com/de-de/aspnet/core/razor-pages/?view=aspnetcore-5.0&tabs=visual-studio. [Zugriff am 29 5 2021].

[7] Microsoft Corporation, „Blazor-Hostingmodelle in ASP.NET Core,“ Microsoft Corporation, 7 12 2020. [Online]. Available: https://docs.microsoft.com/de-de/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0#:~:text=Eine%20Blazor%20Server-App%20basiert,die%20temporäre%20Netzwerkunterbrechungen%20tolerieren%20können.. [Zugriff am 29 5 2021].

[8] Microsoft Corporation, „Entity Framework Core,“ Microsoft Corporation, 20 9 2020. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Zugriff am 5 6 2021].

[9] JetBrains s.r.o., „ReSharper - Die Visual-Studio-Erweiterung für .NET-Entwickler," JetBrains s.r.o., 27 5 2021. [Online]. Available: https://www.jetbrains.com/resharper/. [Zugriff am 27 5 2021].

[10] SonarSource S.A, „SonarQube Documentation," SonarSource S.A, 27 5 2021. [Online]. Available: https://docs.sonarqube.org/latest/. [Zugriff am 27 5 2021].

[11] R. Morris, „Swashbuckle.AspNetCore," Swashbuckle, 18 5 2021. [Online]. Available: https://github.com/domaindrivendev/Swashbuckle.AspNetCore. [Zugriff am 18 5 2021].

[12] SmartBear Software, „OpenAPI Specification," SmartBear Software, 12 5 2021. [Online]. Available: https://swagger.io/specification/. [Zugriff am 12 5 2021].

[13] Microsoft Corporation, „What is DevOps?," Microsoft Corporation, 28 5 2021. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-devops/. [Zugriff am 28 5 2021].

[14] Microsoft Corporation, „Whats is Azure?," Microsoft Corporation, 11 06 2021. [Online]. Available: https://azure.microsoft.com/de-de/overview/what-is-azure/. [Zugriff am 11 06 2021].

[15] Microsoft Corporation, „What is Azure SQL?," Microsoft Corporation, 27 7 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview. [Zugriff am 29 5 2021].

[16] Microsoft Corporation, „App Service overview," Microsoft Corporation, 6 7 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/app-service/overview. [Zugriff am 3 6 2020].

[17] Microsoft Corporation, „An introduction to NuGet," Microsoft Corporation, 24 5 2019. [Online]. Available: https://docs.microsoft.com/en-us/nuget/what-is-nuget. [Zugriff am 17 Juni 2021].

[18] Xunit, „About xUnit.net," xUnit, 6 7 2021. [Online]. Available: https://xunit.net. [Zugriff am 6 7 2021].

[19] M. I. Clarius, „moq," o.A., 12 6 2021. [Online]. Available: https://github.com/moq/moq4#readme. [Zugriff am 12 6 2021].

[20] J. Bogard, „AutoMapper," 25 6 2020. [Online]. Available: https://docs.automapper.org/en/stable/index.html. [Zugriff am 27 5 2021].

[21] Fonticons, Inc., „Font Awesome," Fonticons, Inc., 15 6 2021. [Online]. Available: https://fontawesome.com/. [Zugriff am 15 6 2021].

[22] Microsoft Corporation, „Entwerfen eines Microservicedomänenmodells,“ Microsoft Corporation, 30 1 2020. [Online]. Available: https://docs.microsoft.com/de-de/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model. [Zugriff am 6 7 2021].

[23] t2informatik, „Mock-Objekt – die Attrappe in der Softwareentwicklung,“ t2informatik, 30 6 2021. [Online]. Available: https://t2informatik.de/wissen-kompakt/mock-objekt/. [Zugriff am 30 6 2021].

[24] MathWorks, „Mocking Framework,“ MathWorks, 5 7 2021. [Online]. Available: https://www.mathworks.com/help/matlab/mocking-framework.html. [Zugriff am 5 7 2021].

[25] E. Charbeneau, „10 Blazor Features you Probably Didn´t Know,“ Telerik, 15 12 2020. [Online]. Available: https://www.telerik.com/blogs/10-blazor-features-you-probably-didnt-know. [Zugriff am 6 7 2021].

[26] R. A. Tom Dykstra, „Tutorial: Erste Schritte mit EF Core in einer ASP.NET Core MVC-Web-App,“ Microsoft Corporation, 6 11 2020. [Online]. Available: https://docs.microsoft.com/de-de/aspnet/core/data/ef-mvc/intro?view=aspnetcore-5.0. [Zugriff am 30 06 2021].

Ehrenwörtliche Erklärung

Name:	Vorname:
Herkelmann	Mathias
Matrikel-Nr.:	Studiengang:
752437	SWB

Hiermit versichere ich, Mathias Herkelmann, dass ich die vorliegende Bachelorarbeit mit dem Titel „Entwicklung einer Webanwendung für den Test von Batteriezellen mit ASP.NET und Blazor“ 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.

Ostfildern, 12.07.2021 _____

Ort, Datum Unterschrift